DISSERTATION
Domain-Specific Languages as a Method for Representation and Evolutionary Search Among Global Solution Space of Cognitively Plausible Algorithmic Behaviors

Ryan Kaulakis
Applied Cognitive Science Lab
College of Information Sciences and Technology
rmk216@ist.psu.edu

August 30, 2020

Abstract

This work describes the design and implementation of a Domain-Specific Language Compiler, which permits evolutionary mechanisms to be quantified in terms of the measured behavior of the compiled ACT-R production rules and models that the compiler produces. The method permits the modeling of learning and algorithmic behaviors of arbitrary nesting and complexity. The primary aim of this work is not to solely to learn more about the example application being modeled for this work (i.e. The Block Sorting Task), but instead to prove that the mechanisms can be combined and function together. Ultimately, the goal of this is a long-term research agenda, which applies evolution to modeling real users' algorithmic and strategic behaviors using quantitative models within cognitive architectures. To this end, a nonlinear general regression technique for matching human data with cognitive architectures is introduced. As well as a means of representing both individual behaviors and aggregate behaviors, as well as functions, which operate over these representations. Sequential problem-solving data generated by humans is used to construct programs in ACT-R, which approximate how that data was generated by the original humans. These programs will be constructed with Genetic Programming variants designed to evolve programs in ACT-R that solve the same task as the original human. These heuristically guide accurate matching of the observed data of a single human by maximizing their match percentage over the greatest subset of the human data. The resulting programs can be viewed as approximations of the algorithm used by the human to generate the original data. Further, the results from multiple humans can be aggregated in Program Space, and clustered to produce groups of programs which solve problems in similar ways. These clusters are defined as fuzzy clusters, and referred to as Strategy Groups, because they are groups that approximate some heuristic, which humans use when solving the original problem. Strategy Groups designed to be used to perform several key operations which the original unaggregated data cannot, including sampling, composition, reverse prediction, and verification. Together, this research agenda forms the context in which Domain-Specific Language Compilers are required for efficiency as well as being able to read and interpret the algorithms that they represent for humans they are used to model.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Chapter 1

Introduction

1.1 Summary

This work[1] aims to enable the creation of novel sequences of behavioral primitives. Normal ACT-R learning capabilities are able to model improving the speed, utility, or reliability of existing hand-written production rules, but it is outside of the current scope of ACT-R to significantly recombine or rethread those rules beyond the original hand-written ones. In this work, a layer above and prior to ACT-R is used to evolve novel combinations of primitives, which are turned into executable ACT-R models. When run, these models produce feedback to guide the evolutionary process via fitness metrics. In this case, the highest fitness most closely resembles the behaviors that a specific person being modeled did for the same algorithmic problem-solving task. For the purposes of this work, a novel task has been created. Though not itself the focus of this body of research, but it is instead intended as a representative task, which is sufficiently complex without being overcomplicated, while also being reasonably similar to real tasks. The intention of this work is that a real task could be modeled using the approach and techniques introduced herein. For example, by making automatic models of experts, those models act as stand-ins for those experts. Imagine a scenario where an expert driver or pilot could be replaced with an automated stand-in for testing or validation purposes. This could also be used to find out what Strategy or Strategy Groups the expert is demonstrating when observed, even if they could not fully articulate their thoughts. Such a model would be of interest to anyone in the same field as the expert, for the ability to use a model to stand-in for the limited time of experts. Those interested in the modeling of human cognition would have similar but divergent applications for these models, as a way of providing executable predictive models of the internals of human problem solving.

The diagrams within Figures 1.1 and 1.2 represent these layers, and will be shown in several places in this document for clarity. Essential to this work is the notion that there is something to be gained by having these different layers connected in such as way–namely establishing feedback loops that would otherwise being either difficult or impossible without such a system.

Following Figure 1.1, the interpretation for one of the smallest feedback loops is to look at the nodes labeled "Evolution" and "GEVA"[2], where the arrow labeled "Populations" goes from "Evolution" to "GEVA". This means that the Evolutionary Algorithm runtime gives GEVA a "Population" in the form of a list of integer vectors representing one or more individuals in the population which are each "Integer Chromosome Genotype(s)" from Figure 1.2. While "GEVA" from 1.1 participates in further feedback loops to eventually give "Evolution" back a list of "Fitness Values" which give one "Fitness" value for each individual of the population, usually in the form of non-negative reals, $\mathbb{R}_{\geq 0}$ (e.g. $[0.1, 1000000.0, 2.0, 0.0]$). For our purposes, the highest fitness is 0 where there is no observable difference between our model and the human being modeled.

Moving from left to right in Figure 1.1. The next feedback loop is to the right of "GEVA" towards "Block Sorting DSL Compiler", with the two arrows, "DSL Source Code" and "Fitness Metrics". Fitness Metrics are collections of model traces similar to "Traces From Running Model" from Figure 1.2 as well as the necessary metadata for "GEVA" to calculate a "Fitness" value, as mentioned earlier, generally timing information as well other calculated or measured metrics. "DSL Source Code" in this case is "Block Sorting DSL Source Code Phenotype" from Figure 1.2, where the two arrows going into that box represent the deterministic method by which the Grammatical Evolution algorithm maps integer genotypes to character string phenotypes by way of a the Block Sorting BNF Grammar. It is explained

[1] This work has been prepared in LaTeXspecifically for viewing as PDF document. When viewed as a PDF, all citations and references in this document are clickable hyperlinks, and will permit you to jump to the page of the link's target. The Bibliography contains back references to all citations of each bibliographic item. Other links are one-way. This should help facilitate reader navigation.

[2] GEVA is the name of a Grammatical Evolution library used in this work.

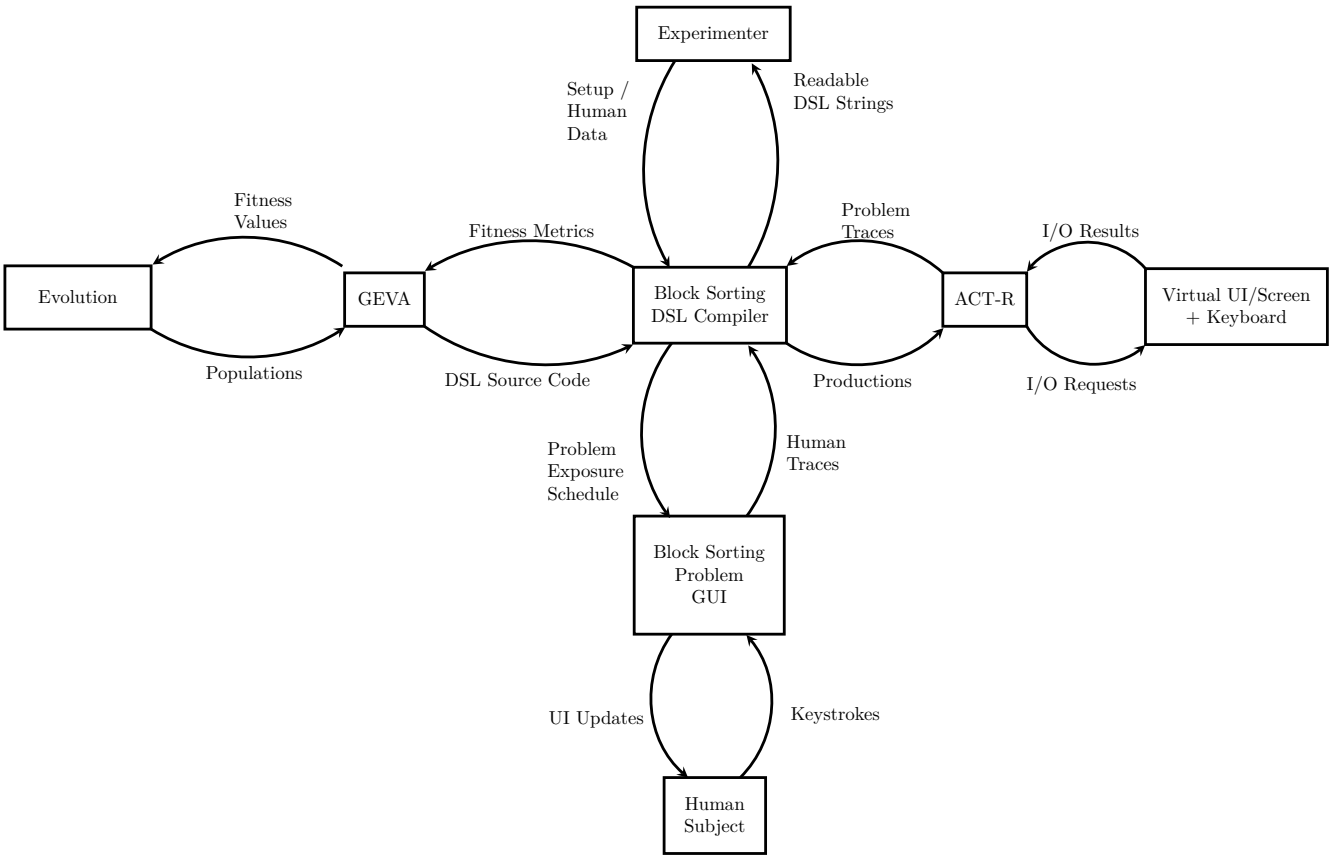

Figure 1.1: Thesis Layers Diagram

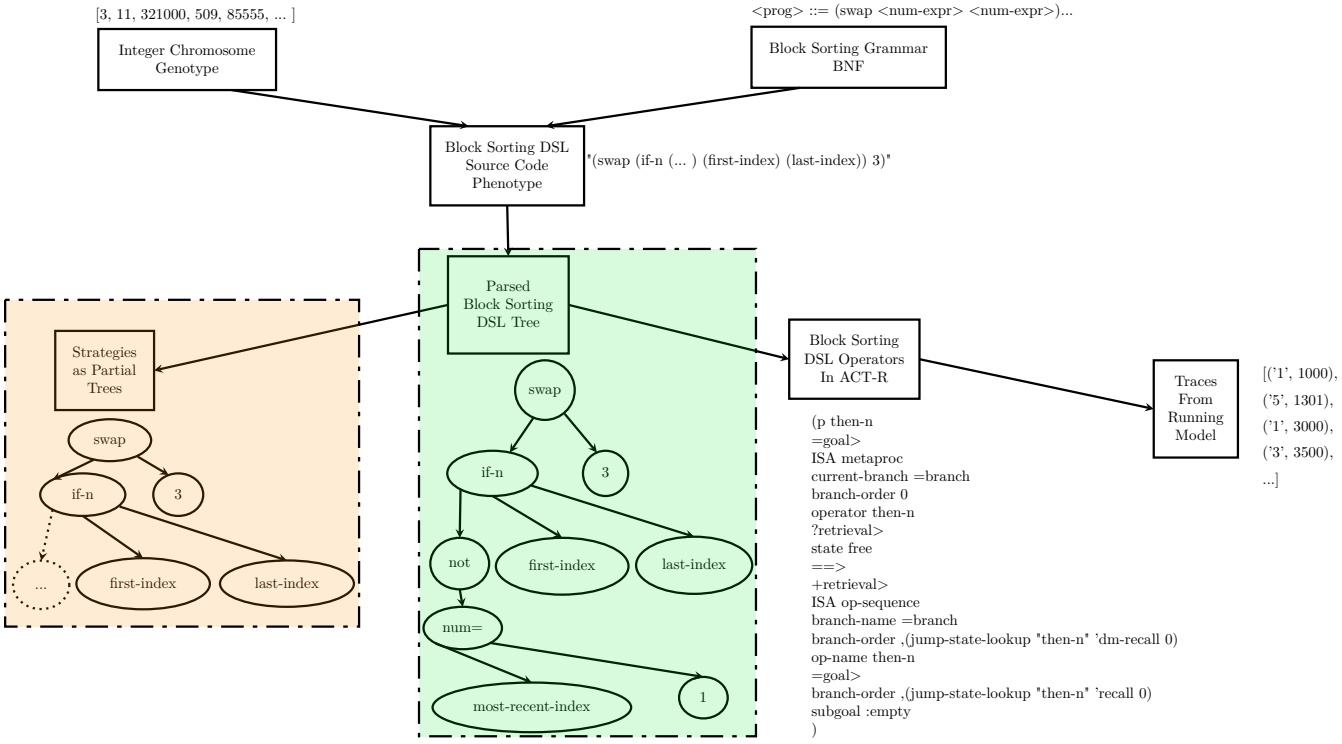

Figure 1.2: Thesis Representations Diagram

in greater detail elsewhere, but the key is that it makes only grammatically legal programs for our Block Sorting DSL, and it makes them as character strings, just like a hand-written program would be. The program string is fed as input data to the "Block Sorting DSL Compiler" where it parses it into "Parsed Block Sorting DSL Tree" before compiling it into "Block Sorting DSL Operators In ACT-R" in Figure 1.2. As the implementation of the Block Sorting DSL Grammar, the compiler is where most of the experimental theory is held. In this particular feedback loop (i.e. Figure 1.1), it acts to take an input individual from the "Evolution" node, expanded into a program to test by "GEVA", and then after the model is run it passes that information back up to "GEVA".

Directly following on from that step, Figure 1.1 next feedback loop from left to right is between "Block Sorting DSL Compiler" and "ACT-R", where that "Block Sorting DSL Operators In ACT-R" in Figure 1.2 is passed in the "Productions" arrow in Figure 1.1. This is no different than having "ACT-R" run a normal hand-written model, save that the models are instrumented with hooks for timers and loggers in a way that does not change the runtime behavior of the models, but which just handles when models start and stop. The reverse arrow, "Problem Traces" has been mentioned in passing as part of the "Fitness Metrics", but they are simple keyboard keystroke logs, recording when specific number row keys were pressed in milliseconds, as seen in Figure 1.2 as "Traces From Running Model". It is the case that "ACT-R" is run within an experimental harness, which sets up a fresh "ACT-R" instance (as well as a fresh Common Lisp instance) for each model that is being evaluated, thus ensuring that there is no chance for models to interfere with one another.

The final, and most grounded feedback loop on this axis of Figure 1.1 is the one between "ACT-R" and "Virtual UI/Screen + Keyboard". The two arrows, "I/O Requests" and "I/O Results" logically represent the "ACT-R" Motor Module attempting to strike keys that the "Evolution" individual requested it to hit, while the "ACT-R" Visual and Aural systems receive feedback from the "Virtual UI". This Virtual UI is almost identical to the normal experimental UI in "Block Sorting Problem GUI", save that it is implemented in Tcl/Tk and may be user-visible or not, while remaining visible to the ACT-R runtime and the model it is running[3]. The key here is that models may attempt to do all kinds of keystrokes, but not all of them will cause changes in the state of the UI. The "ACT-R" experimental harness manages this "Virtual UI", as well as instruments it in order to produce keystroke logs which are cleaned up to make "Problem Traces" from Figure 1.2. These are created fresh for each "ACT-R" instance, as part of that experimental harness.

Reorienting from the left-to-right axis of Figure 1.1 to the vertical one, the logical starting point is the feedback loop between the "Experimenter" and "Block Sorting DSL Compiler". Here, the "Experimenter" gives "Setup / Human Data" to the DSL Compiler, and they get back "Readable DSL Strings" back from it. These "Readable DSL Strings" are "Block Sorting DSL Source Code Phenotype" as seen in Figure 1.2 while the representation of the "Human Data" is the same as "Traces From Running Model" from Figure 1.2. This should be understood such that humans are measured in as identical ways as models, so as to keep the experimental differences between them isolated to what mental logic was guiding the human or model while it was acting to solve the problems. As well, the "Experimenter" determines many details of the experimental harness as they perform the "Setup"; most of these are either ACT-R parameters, or timers, or fitness coefficients. The "Experimenter" is assumed to be specifically interested in the DSL program strings as they are the primary output of this entire modeling process. Each program string represents for the experimenter, an executable theory of mind. This executable theory models a specific person's mental behavior while solving problems in the domain of interest. These can be read like normal source code, but the really unique use is to use Tree Difference techniques to extract common subsets of the individual programs, which are therefore common algorithmic templates that the problem causes people and models to use to solve it, per Figure 1.2 "Strategies as Partial Trees". For example, if every person in the study employs behaviors which visually check the location of the letter *A* at the start of each new problem, that common behavior can be detected and represented in this way.

Moving down the vertical axis of Figure 1.1 the next feedback loop is between "Block Sorting DSL Compiler" and "Block Sorting Problem GUI". Here the "Human Traces" are in the same format as the "Human Data" previously mentioned, and the DSL runtime is being used to administer the experiment that the "Block Sorting Problem GUI" is presenting to a human test subject, almost identically as how "ACT-R" and the DSL runtime did for models on the left-to-right axis earlier. As well, the source of the "Human Data" that the "Experimenter" gives to the DSL Compiler is almost always (unless they provide their own) from earlier recordings from this feedback loop. Whereas the other parts have at least some prior mention, the "Problem Exposure Schedule" has nothing earlier, aside from being part of the "Setup" implicitly. What it actually does is take instructions from the "Experimenter" about how many problems there should be, how long they should be, how many *repeaters* there should be, and how often a *repeater* problem should be repeated. While most of these are pretty obvious, the *repeaters* are randomly generated like everything else but are inserted in the testing schedule for the express purpose of measuring expertise over the course of the experimental schedule. The premise behind them is that among all the random problems, they are the

[3]The host systems may change things like absolute line widths and font sizes, but the overall UI designs are otherwise identical, and the differences were not noticeable to most pilot subjects, when asked.

same problem with the same order and complexity shown and solved many times over the course of the experiment. Without them, modeling expertise would be harder.

The final feedback loop in this axis of Figure 1.1 (and in an absolute sense) is between "Block Sorting Problem GUI" and the "Human Subject". It is almost identical to the one between "ACT-R" and "Virtual UI/Screen + Keyboard", save for the fact that the GUI is a real visible UI on a real screen, and the keyboard is a real physical keyboard. The measurement systems are in Java, as is the UI, but that changes little besides the font. All of the same feedback is provided and recorded between the human and the UI as would happen between the model and its own UI.

Most of these feedback loops and their representational products can be used for a variety of analytical tasks, as detailed in Figure 3.1. They are divided between Bootstrapping Tasks (above the dashed line)–which create seed populations, human data, and model data–and Post-Process Tasks (below the dashed line), which use those products to guide further inquiry or as outputs themselves.

1.2 Premise

There is a desire to create computational models of people solving tasks. Current methods utilize computational models that generate simulations of human data, as if a human was doing the same task. These methods work well enough for some tasks, but the process is a highly manual one. There are no guarantees that the resulting models represent the range of possible algorithms, which people use for that task. More likely, the experimenter would need to make many separate models for the same task, each one a separate executable theory of how the person solves the problem. Such work is not often done, for practical reasons.

It is the desire to automate this process of theory generation (in the form of a model of behavior), testing (in the form of behavioral data from humans and models), and revision that motivates this work. With the end goal being the creation of a research methodology not only performs these operations, but is backed by a theory that links the methodology with the exploration of clusters in Program Space[4].

The impetus driving this method is that–in cases where there is no single canonical algorithm to solve a problem (which are many)– there are plenty of viable behaviors that we would have no sensible way of talking about or representing computationally because we lack a theory to describe them. Furthermore, attempts to do this kind of modeling without the methodology this work describes would run into problems that would necessitate the creation of a system similar to the one described in this work.

To demarcate the bounds of this work, this dissertation describes the foundations for what is expected to be more than a decade worth of follow-on work, and thus is part of an extended research agenda. Building on specific experimental attempts, this work addresses all of the previous concerns while still leaving the open question of what ultimate results may be discovered when running the system for detailed experimental work (perhaps for weeks on conventional hardware, or days on HPC systems). This work draws a line at this point, and its scope encompasses the fundamental designs necessary for that final time and resource-intensive exploration to happen.

There is no clear measure of the size of the space to be explored by the evolutionary processes. There are no hard guarantees that interesting results could be produced in a timely fashion, only the strength of our heuristics. In contrast to that point of view, this work is worthy of being reported independently, as it contains both insights and novel contributions all its own. Instead, this agenda is presented here to contextualize this work and the unique requirements that informed its design; the novel work here stands on its own, but it is more easily understood with added information.

Within this scope, human pilot experiments were used to collect real data for an example algorithmic task, The Block Sorting Task, and a real Domain-Specific Language (DSL) was designed and implemented using a custom DSL Compiler which targets the ACT-R Cognitive Architecture. As well, a Evolutionary Algorithm was designed to target this DSL and evaluate the fitness for purpose of randomly generated programs in this DSL, which would be turned into quantitatively predictive real executable ACT-R models. For this to work, a comprehensive generic model of arbitrarily nested complex behavior was designed, implementing a Von Neumann Architecture within ACT-R; any weaker model would be insufficiently powerful to handle representing the multifarious range of viable human algorithmic behaviors for The Block Sorting Task.

1.3 Component Introduction

Given this high-level view, it is appropriate to drill down and examine the impetus behind each part and provide forward-references to sections of this dissertation. As a whole, the work is complex and multifaceted, but the individual

[4]See Section 2.3.3 for more details on Program Space.

parts can be understood in relative isolation.

Beginning with the core task of this research, the Block Sorting Task, it is important to understand that the focus of this research is on quantitative modeling of individual algorithmic behavior in humans. For this end, there needed to be some exemplar task to use to demonstrate the basic premise of the work. Though other illustrative tasks are possible, The Block Sorting Task was selected because that there are both psychological descriptions of human seriation behavior as well as algorithmic sorting from computer science.

Instead of directly copying another task, a simplified version was used[5] which reduced the complexity of seriation tasks by removing the hidden weight component of Gascon (1976), while also modifying the version from Young (1976) to work better with a computer interface. The Block Sorting Task was devised in place of those, which is intrinsically computer-moderated and replaces balanced-beams with pairwise swapping only. Basically, a person is presented with a user interface screen which represents alphabet blocks, like children might use for a normal seriation task. Here, however, they are always out of order at the beginning and the experimental subject must strike the number-row keys representing the pair of numbered slots they wish the interface to swap in place. When the blocks are finally sorted, a tone plays and the next problem is presented in the problem set for that experiment, unless the problem set is done. See Section 3.1 for more details.

Moving on, the next logical place to focus on, is the Domain-Specific Language part of this work. Given the Block Sorting Task, experimentation eventually lead the author to conclude that it was reasonable to try to represent the range of possible behaviors for the Block Sorting Task. Instead of directly grounding discussion in some cognitive architecture, this work approaches it from the angle of trying to capture the domain of sorting blocks–explicitly within the task environment and UI. Within the programming language research community, there is a category of programming languages called a Domain-Specific Language (abbreviated DSL), that is specifically designed to model specific domains rather than general programming tasks.

To represent the knowledge and structure of the domain of sorting blocks, this work created the Block Sorting Grammar, which is an Embedded DSL. For this purpose, a Backus-Naur Form[6] grammar (Backus, 1959; Naur, 1963) was created to reify the Block Sorting Grammar. This is discussed extensively in Section 3.2. The main take-away of which is that there are logical behavioral options available to a experimental subject, which are codified in Operators within the DSL. Operators following their BNF grammar have rules about how they can be composed and combined, and by design are capable of arbitrarily complex nestings, so long as they are grammatically valid.

As an Embedded DSL, the actual implementation of the Block Sorting DSL Compiler is grounded within the ACT-R Cognitive Architecture (Anderson, 1990; Anderson et al., 2004; Anderson, 2009) and its underlying Common Lisp runtime. The details of this compiler form the largest component of this dissertation, which is discussed at length beginning in Section 3.1. Without going into the details yet, at this point, the key detail is that the final use of the compiler is this: a researcher doing cognitive modeling for an algorithmic task could write down a program in a DSL for that task, and only need to concern themselves with the high-level details of describing the behavior. Meanwhile, the DSL Compiler creates executable quantitative models from whatever behavior that researcher's DSL program describes. It not only permits a high-level view of modeling, but it also automatically creates executable testable quantitative predictions about how plausible that model might be.

Finally, the portion of research which informs all of this design, and which ultimately sets the stage for follow-on work, this whole system was designed to be automated. Building upon the idea that any grammatically legal program in the Block Sorting DSL would be an executable testable quantitative theory about how an individual might attempt to solve the Block Sorting Task, the automation of this is done through the use of Evolution. Specifically, the kind that evolves grammatically constrained populations of programs, using Grammatical Evolution from O'Neill and Ryan (2001).

In this formulation, which forms the context for future work, now that the DSL Compiler is completed, the Evolutionary Algorithm randomly generates Block Sorting DSL programs. These are then evaluated against human data for the same problems, causing the fitness to favor behaviors which produce similar keystroke logs as the individual being modeled produced. After breeding these programs for a user-determined period, increasingly fit models of that individual's behaviors are produced. These models are rather unique, in that they are simultaneously readable by the experimenter without any statistical knowledge, as well as being mutually comparable using Tree Difference algorithms to find how similar two programs are structurally.

[5] Suggested by the committee members.

[6] Abbreviated BNF.

1.4 Experimental Evaluation

This work takes human data from problem solving tasks, and uses Evolutionary Algorithms to generate a population of programs designed to mimic the process which created the human data. Special constraints are used to make this task feasible, beginning with the programs being grounded in a cognitive architecture, ACT-R[7], which itself is used to generate data which predicts human data. Since the ACT-R theory says that it attempts to mimic the architecture of a human mind, then these programs or models are executable theories of how people do tasks. Furthermore, the programs generated are represented in a Domain-Specific Language which captures the range of possible operations and datum of the problem task being solved, and ultimately translates them into ACT-R production rules.

Finally, the Genetic Programming variant system utilizes a special fitness heuristic in order to match the data. A particular person is used to generate several sequential behaviors which end with the solution of a particular problem. The sequential data consists of time-stamps, the problem state, a person's choice, and the result of that choice are all collected into a log called a Trace. This person generates several traces, the heuristic then tries to maximize the percentage match of all of the Traces. This match is based off of a comparison with to simulated Traces generated by programs from the Genetic Programming population which are run on the same problems (usually in the same order as the person). A single program will generate one simulated Trace for each real Trace, and its fitness will be the sum of the percentage matches of each Trace pair. While a single pair of traces will be compared using a weighted sum of longest common sub-sequence, and temporal distance between the human Trace and the program Trace for the common parts (the sub-sequences and select reference points of the problem, e.g. start or end). The Genetic Program runs either until sufficiently many good matches are found or until some number of generations have passed. The results of this process will be a set of programs which approximate the behaviors seen in a particular person.

When the results of applying the Genetic Programming process to several individuals are collected, the result will be a set of program populations which all mimic particular people[8]. The experimenter could stop here and simply examine the programs to gain insight into what script the human was using to produce their data. However, the real use of these populations is to use them to explore the Program Space, and then see where high fitness clusters of programs occur in it. By applying a suitable clustering method, programs will be grouped by physical similarity, which can allow analysis methods to extract the similar structures from the clusters that represent the shared component of the programs in the cluster as a heuristic. These clusters this work refers to as Strategy Groups.

The expected contribution of this research agenda is a methodology for working with Strategy Groups. Both at the individual level, where the variability of individuals is captured and described, as well as at the aggregate level, where individuals are clustered by the programs that represent them to form Strategy Groups. These groups function analogously to probability distributions, and permit statistic analysis and sampling. When this methodology is fully developed, it is expected to open up entire new vistas for exploration via computational cognitive modeling.

For more details see Section 3.1, which includes how to create more programs, preseeding guidelines, clustering details, and other potentially interesting information about this process.

1.5 Example Applications

In this section are included three high-level example applications of the regression method which this research aims to support, at both the level of individual modeling as well as the level of aggregate modeling. Each of these examples has been chosen based on a connection to prior work in the field of Cognitive Modeling. To make these examples both solid and easy to understand, the details of the techniques will be elided (though they can be found in Sections 3.1 and 5.1).

1.5.1 Intelligent Tutoring Systems

Computer-controlled tutoring systems have existed for decades. Intelligent Tutoring Systems (ITS) one of the more interesting types of these to this work. They tend to include theories about learning and memory as those things apply to a student learning a specific subject (Sleeman, Brown, et al., 1982; Corbett, Koedinger, & Anderson, 1997). A difficulty that an ITS system designer will face is that it is difficult to figure out what a student has learned based on what they are doing. As well, the range of behaviors that they need to anticipate in the students is also not normally known.

If a ITS designer were to apply this work's regression method to their problem, they would have two different levels of benefits. First, if they apply the individual-level operations, they will be able to automatically create models

[7] This work uses ACT-R 6, but should apply equally to later versions.

[8] While this work aims to support this kind of future work, the scope of this dissertation is strictly about how to characterize an individual computationally, such that this future work can be pursued.

which accurately approximate specific students. Such models would be based on the data traces collected from a particular student solving several related problems. The researcher could then either use the models as a predictor for that student's future attempts at solving problems. Additionally they could inspect the models themselves to see what rules the student is probably using that make them act in the ways observed, perhaps to get feedback to shape future corrective training.

When several students are available for testing, the researcher is able to utilize the aggregate-level operations from my regression method, by combining the results of the applying the individual-level operators to each individual student. By doing this, my regression method would form Strategy Groups which allow the researcher to classify the behavior of multiple individuals as being generated by similar thoughts. For example, common errors in understanding could be represented in this way. Also, the Strategy Groups themselves help to demonstrate the range of possible behaviors that students might show when using the ITS system. This information would include both the kind of behaviors, and the relative frequency of them that might be expected from the population (i.e. the classroom, not necessarily the general populace).

1.5.2 HCI Interface Design

Human-Computer Interfaces (HCI) considers the design and testing of interfaces as one of its main foci. While there are many methods of representing interactions with interfaces such as GOMS (Card, Moran, & Newell, 1986), the designer is still left with the task of figuring out how a end user will utilize that interface. Based on these assumptions, the designer may change their interface to make it more productive. Some studies show that such changes can save large amounts of money such as (Panel on Modeling Human Behavior and Command Decision Making: Representations for Military Simulations and National Research Council, 1998).

When a HCI designer needs to test an interface, it is helpful to have an automated model of a user to test the interface with (Ritter, 1993a, 1993b; Ritter & Larkin, 1994; Wallach, Fackert, & Albach, 2019). Making such models is normally nontrivial. If this work's regression method is used, the designer would be able to create models of individual users, including people with behavior representative of key groups of interest (such as novice elderly users, or expert female users in their thirties). Besides simply being able to test the interfaces using the models of these users, the HCI designer can inspect the models themselves to discern what rules are being used by the people being modeled. This may shine light on behaviors that could otherwise be opaque, and thus ultimately inform design of interfaces. For example, an HCI designer could read model rules, and then use those rules to inform which parts of an UI receive hints or interface clues to help direct attention to where the models predict the person is likely to need to attend to next. Experts are notoriously bad at explaining themselves, but this method creates iteratively refined models of those experts using the interface in question, which can be read clearly by a researcher or reused. Following the prior example, the model of the expert could guide non-experts as well, by hinting at what the expert is likely to do next.

A greater benefit can be found in the aggregate level regression operations. With many user models being utilized, the designer can interpret the Strategy Groups as representing interface-usage heuristics that are common to both the wider population, as well as their specific interface. Even relative frequency of strategy occurrences can be used to predict how users will utilize an interface. As a result, the designer can optimize for their preferred strategies by making it more efficient to use, and by giving cues to encourage most end users to use the designer's preferred strategy.

1.5.3 Navigational Tank Control

Drawing on the example of a Subject Matter Expert (SME) labeling sections of a map correctly (Best & Gerhart, 2011) for a tank to drive on, the key issues that need to be modeled are correct labeling given circumstances, correct intermediate steps (like eye-tracking attendance to features), and accurate timing. In this task, the expert drives a tank in the dTank simulator and competes against an opponent tank commander in a 2D environment which features obstacles. The SME labels certain areas of the map as being high to low risk areas, while the visual focus and other timings are recorded.

Should an experimenter wish to analyze their subjects using this work's regression method, they will need to input the parallel timings of both the interface usage by the SME as well as the eye-tracker timings and placements[9]. The system will output a set of programs which model the specific behaviors shown by that SME, which could either be used to explain how the expert acts, or to act as a stand-in opponent for other SME's.

With programs modeling multiple SME's, the aggregate level of the regression method allows them to be categorized and classified. Thus, general heuristics used by the SME's can be represented, and extracted for later use. Further,

[9]This application would need to wait for the generalized parallel factor version of my regression method to be ready. It is described alongside the normal version in this paper, but testing isn't planned until the proof of concept serial only version has been tested.

the regression method is capable of predicting new behavior outside of the range of the sampled experts. If it were used for a real military training application, its predictions of strategies unused by current experts may either be helpful for training against unmet novel opponents, or for suggesting new heuristics that the experts themselves may be able to utilize.

1.6 Research Timeline

Based on committee feedback during the initial proposal, it was estimated that the scope of the fully explored research space, including the Strategy Groups, would consist of a decade or more of research. In order to have a reportable milestone, this work presents the first working DSL Compiler rather than the eventual problem application and result reporting.

Later work in this research agenda is intended to be published on its own. This will include reporting actual results of evolving high-significance models of individuals, as well as future refinements of the DSL Designs to permit models of other tasks besides the exemplar Block Sorting Task.

However, the current status of the research is detailed below. Almost all parts of the work have been completed and used extensively. Except as detailed at the end of this section, all of Figure 1.1 are all tested and working. Of these parts, the DSL Compiler has been used to generate thousands of test programs successfully. As well, all of the experimental apparatus that interacted with human subjects are all fully working and have been used on real humans without issue. Most portions have been not only unit tested in isolation, but also integration tested as whole working parts of Figure 1.1.

The following parts have only been unit tested, meaning that their basic functionality has been validated against test problems. They are the portions of Figure 1.1 on the right-to-left arrows from "Block Sorting DSL Compiler" to "GEVA" to "Evolution". Each portion has been unit tested in isolation, but not necessarily with real human experimental data. For example, a test implementation with a modified version of the Bubble Sort algorithm (Friend, 1956; Knuth, 1997). With several of the earlier BNF grammars, hand-encoding was performed, resulting in manual selection of integer genes in the genotype which the Grammatical Evolution process expanded into early Block Sorting DSL programs which performed a manual version of the essential pairwise comparison and bubbling behavior from the Bubble Sort algorithm, but which largely ignored the added visual knowledge that the normal algorithm does not normally have access to but which a human does; these manual genotypes are tied entirely to the details of the specific BNF used, as each BNF fully determines a different DSL Language. Another example can be seen in unit testing each individual Operator and the arbitrary depth limits for their nesting using extensive coverage with `ASSERT-N`, `ASSERT-C`, and `ASSERT-B` as detailed in Section 3.4.33.

The functionality involving Strategy Groups (the lower left of Figure 1.2) and the analysis that they support is strictly theoretical at this point. Lacking real large populations of DSL programs that already mimic actual humans, to do analysis with, the processes involved merely serve to inform the design of this work to support some kind of experimental design. If that was not a design consideration, it would free some design constraints from this work.

1.7 Layout

This work is divided into three parts. The first part, in chapter one delineate the background information, and literature review. The second part consists of chapter two, and contains the current state of the methodology of this work. Finally chapter three and onward are research results various kinds, including the bulk of the novel work. There are also appendices after the bibliography.

1.8 Contributions Summary

This section provides a concise summary of the novel contributions of this work. A expanded version can be found in Section 4.2 at the end of this dissertation. The description here is sufficient for browsing without a detailed reading of the remainder of the dissertation, whereas the later presentation makes use of terminology and conceptual references that are explained throughout this document.

1. Grammatical Evolution of Cognitive Models: This is the first work which applies any form of Genetic Programming to the generations of programs representing cognitive models of algorithmic behaviors.

2. Computer-Moderated Adult Block Sorting Task: This work builds upon modeling of tasks for young children with physical objects, and turns it into a reusable computer-moderated task for adults. This work created and implemented this task.

3. Block Sorting Grammar DSL[10]: This work introduces the concept of not only organizing behaviors in terms of task-specific operators, but it also introduces a novel BNF Grammar to formally represent the full range of behaviors possible. This DSL representation is integrated into quantitative and executable models in this work.

4. DSL Compiler: The DSL Compiler and its method of action introduced in this work are new, representing the first of a class of special-purpose tools to reify DSL programs into cognitive architecture programs. In this work, the exemplar is the Block Sorting Grammar DSL, and the target cognitive architecture is ACT-R.

5. Von Neumann Architecture in ACT-R: This work creates the first detailed representation of a Von Neumann Architecture grounded in ACT-R. While this was not the original aim of the work, the author realized that nothing less powerful than a stored program computational architecture would be sufficiently powerful enough to implement the whole range of possible behaviors. As those behaviors not only needed to represent the Block Sorting Grammar DSL, as well as the threading of control needed to execute these experiments, but they also needed to be realizable in ACT-R.

6. Representation of Arbitrarily Complex Nested Behavior: This is the first time that arbitrarily complex nested behavior is permitted in ACT-R without requiring explicit hand-coding of the complete range of programs. It was this design goal that drove the adoption of the DSL, not the other way around. Indeed, it is absolutely required for the used of Evolutionary Algorithms to fit these models.

7. Automatic Individual Modeling: This is the first time that cognitive models of individuals can be automatically generated. While other automated modeling methods have existed, none have been able to create and explore the whole space of possible programs to represent an individual, only in narrower senses of fitting preexisting models. By inference, the stored program would be logically indistinguishable from a hand-coded one, but which is likely to be more complex to implement by hand (e.g. in ACT-R) as well as being potentially more error prone (Ritter et al., 2006).

8. Nuanced Chunking: This work has revealed that the current chunking behavior in ACT-R is not transparent enough to handle a Von Neumann Architecture, and that more nuanced representation may be required. If an amended chunking algorithm were used, the parts of the algorithmic behavior which are constant would be able to benefit from chunking, where the current chunking algorithm does not.

[10] Domain-Specific Language

Chapter 2

Background Review

2.1 General Information

The information in this section serves to contextualize the foundation upon which this research is built. Additional commentary is provided as well, to link the ideas here to this work.

2.2 Cognitive Architectures

For the purposes of this paper, Cognitive Architectures are defined as unified computational theories of human behavior (Newell, 1972, 1990). The impetus behind them is simple, to quote Allen Newell[1]:

> The question for me is, how can the human mind occur in the physical universe? We now know that the world is governed by physics. We now understand the way biology nestles comfortably within that. The issue is, how will the mind do that as well? The answer must have the details. I have got to know how the gears clank and how the pistons go and all the rest of that detail. My question leads me down to worry about the architecture.

Cognitive Architectures then, are the theories about those details. Generally, they are theories embodied in a set of principles and equations which are then developed into some form of executable software[2]. The unification in this case attempts to limit the range of valid architectures to those which can support many different human behaviors simultaneously, rather than having one purpose built architecture for each and every individual human behavior. For example, instead of individual architectures for playing poker and another for playing blackjack, there should be one that can handle examining cards, reading rules, and making game-play decisions which could be run on either poker or blackjack, as well as any other representable card game. Ideally, this would represent the whole range of human behaviors in a single unified architecture, but the best way to do this is very much an open question of the field. Some architectures dig deeper than others into the physical concrete implementation connecting thoughts and behaviors down into neurons and chemistry, but all are theoretically grounded in the real world, see Section 2.7 for more details.

As was stated earlier, all Cognitive Architectures are unified theories of mind. As one of the founders of Cognitive Architecture research, Allen Newell had long argued that only a unified theory of mind could be tenable, from his early paper (Newell, 1972) famously claiming "You can't play 20-questions with Nature and win.", to his ultimate attempt at implementing a unified theory of cognition. This attempt was embodied in the titular book (Newell, 1990) and the related Soar Architecture (Laird, Newell, & Rosenbloom, 1987). His ideas have spread, as all modern Cognitive Architectures that this author is aware of each represent some kind of unified theory—though not necessarily all the same unified theory. Even if it has gaps in coverage, the concept that it should be unified serves only to drive research to fill them.

How "the gears clank" differs from system to system. Each cognitive architecture seems to reflect the individual research and preferences of their authors, packaging together their own set of theoretical commitments and functionality together. These differences can be significant, for example the EPIC Cognitive Architecture claims (Kieras & Meyer,

[1] His "Desires and Diversions" farewell speech before dying from cancer in 1992, video published in 1993. The "question" he is referring to in this quote is explained in context as one of the "ultimate scientific questions" which some people are lucky enough to find themselves captivated with. The "question" in question is thus the motive for his life's work.

[2] Though the theory itself is normally referred to as the architecture because it describes a system by analogy with how normal architecture describes a building.

1997) that humans are innately parallel, and capable of performing multiple simultaneous tasks. While the ACT-R Cognitive Architecture claims (Anderson, 1990) that there exists a cognitive bottleneck, which forces all tasks to run serially. Unless particular care is taken when choosing an architecture for some application, these differing theories may result in different predictions. It is advisable to choose one based on the backing theory in addition to any practical considerations.

There are a number of architectures to choose from, developed by many different researchers and under many different theories. For example, current candidates for this author's future research include: ACT-R (Anderson, 1990; Anderson et al., 2004; Anderson, 2009), Soar (Newell, 1990; Laird et al., 1987), and CLARION (Sun, 2001). For this work, ACT-R was chosen as the exemplar for this research agenda, for reasons discussed in Section 2.2.1. Other architectures are options as well, but targeting one is an extensive effort, even with this work to build upon.

Regardless of the architecture chosen, when utilizing an architecture, a model of some task must be made in the framework it provides. Then, the architecture will make predictions about the way humans would behave as they do the same task. Typically, the model is made in the form of a program in a Domain-Specific Language describing the mental processes involved, and each of these models is then an executable theory of human problem solving from some particular task.

The kinds of tasks that are reasonable for inclusion in this eventual line of research are discussed in Section 5.1.5. Architectures have been used for a long time for *process models*[3], which are the kind of tasks I am considering, having grown out of research involving them. Logically, much of this work begins with the theories of Information Processing from A., Shaw, and Simon (1958); Newell and Simon (1972), as a basis for what parts a model of a human solving a problem must have, an enumeration of variables. From there, certain parts are held as constants, and the architectures arose as formalizations of the machinery.

Of particular interest to this work is the ability to apply automation to modeling tasks. While there is a long history of attempts to use computers to handle parts of the modeling process outside of simply running the model, the results of these attempts have seen limited success or adoption. Some examples include:

- Automating testing for protocol analysis (Ritter, 1992).
- Modeling individual differences using shared rules (Miwa & Simon, 1993).
- User modeling (Rauterberg, 1993).
- Human error modeling (Rauterberg, 1995).

2.2.1 ACT-R

Of the Cognitive Architectures examined in preparation for this work, ACT-R is the one most promising for this purpose. It can be taken as representative of what Cognitive Architectures look like up close.

ACT-R is the theory of the *Adaptive Control of Thought-Rational*. It is a modern architecture with a long history of good work behind it by Anderson (1990; 2004; 2009). It particularly has good support for predictions about human memory and attention. For instance, a group at the Penn State Applied Cognitive Science Laboratory, including myself, have used it to test the effects of memory as a constraint on social network models (Zhao et al., 2015). This research uses ACT-R to model real human constraints on social networks. For example, on sites like Facebook or MySpace, people may have lists of thousands of friends. However, we understand that there is only a select number of people that a person really knows well, while the rest are not easily recalled. By analogy, when social network models are being designed, they may be permitting a node to maintain too many social links to be cognitively plausible.

The theory put forth by ACT-R can be summarized thus: the human mind is modular, and communications between modules happen only through a central process using shared memory buffers, which is analogous to passing messages through mailboxes. Looking a Figure 2.1, you can see the orange central area labeled "Production System" as the central process, and the white-colored buffer areas linking to it as the buffers described above. All communication is done through this central system adding or removing things from buffers. Each buffer is specific to a specialized module (the pink or blue areas), which handle any and all functionality of a particular type. The Manual Module, for instance, handles the perception of where and how a person's hands type on the keyboard, but it also handles manual output in the form of performing keystrokes. Another example is the Declarative Module, which stores Long Term Memory elements; commands to its buffer tend to either add new facts about the world, or interrogate the available memories to find if some fact is currently known. Of special interest is the Goal Module, which contains the current information about the goal directing the person's current behavior.

[3]Terminology due to Ritter (1992), where they are defined as "these models predict the sequence of steps a human executes while performing the task.", making them related to the Traces discussed in Section 3.1.

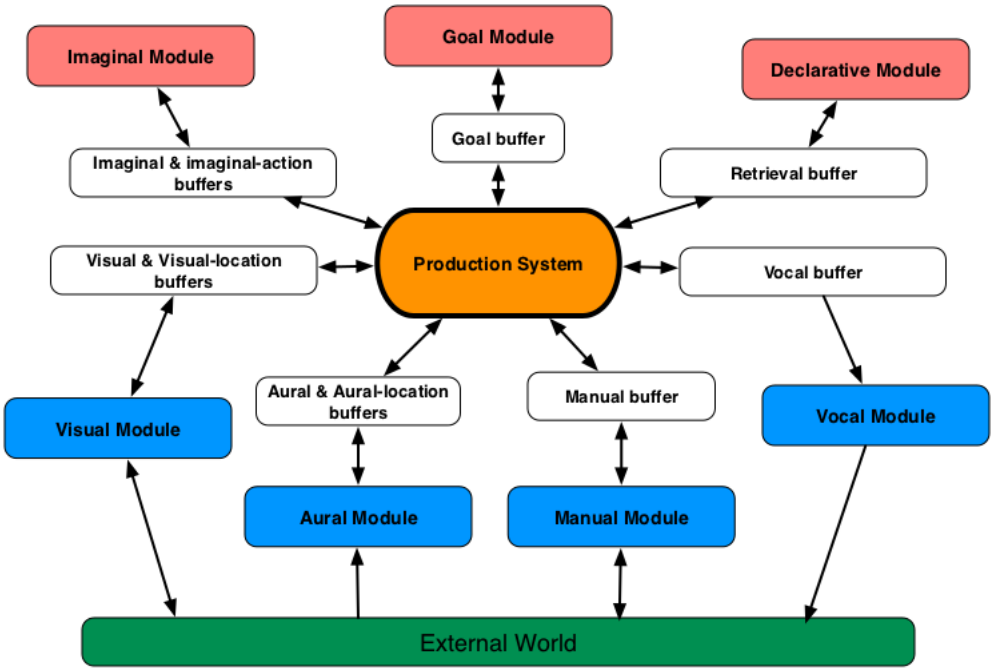

Figure 2.1: Modular Layout of ACT-R 6 (courtesy of Paik and Ritter)

In this work, the orange "Production System" from Figure 2.1 is of great importance. When making a model in ACT-R, a set of if-then rules called *productions* are put in there (and other content is added to Declarative Memory and anywhere else it needs to be, like visual or aural inputs). When apply Genetic Programming techniques to this system, the fact that a simple set of if-then rules is ACT-R's core method of generating behavior means that this system can easily target these if-then rules using GP techniques. While this work does not directly breed productions, it does produce them from domain-specific operators, and then runs those programs to make behavior traces that are then compared to human behavior traces in the fitness evaluation.

Another interesting property of ACT-R models is that they produce very neat and detailed behavioral traces, which closely mimic human data. Any ACT-R program automatically runs through its operations, printing out the sequence of steps and timing that went into them. Such timing data produces a series of timestamped actions which constitute a solid behavior trace similar to a human. Additionally, ACT-R predicts activation levels for particular regions of the brain associated with particular modules, and so its predictions can be tested against human subject's Functional Magnetic Resonance Imaging (fMRI) data. Taken together, these two connections to human data allow for the search space of an GP system trying to match them to be more highly constrained, and thus reduce the total search space being evaluated.

A practical property of ACT-R[4] is that its run-time is highly extensible. Thus, it is relatively easy to install testing apparatus and monitoring functionality into the system for experimentation. Other systems offer their own facilities, but being able to inspect the system from the experimental harness during run-time has proven useful to this work.

Additionally, prior work applying Evolutionary Algorithms to ACT-R exists (Ritter, Kase, Klein, Bennet, & Schoelles, 2017). While the application involved techniques and goals not directly related to this research, it is still the only publication that has done anything like this to date. Instead of applying GP to modify rules, Kase's Doctoral Dissertation (Kase, 2008) (the latest publications of which are (Ritter et al., 2017) and (Ritter, Tehranchi, Dancy, & Kase, in press)) applied a Parallel GA to ACT-R instances to create *overlays* that matched the behavior of human subjects on a stressful Serial Subtraction task both with and without caffeine. Her *overlays* are qualitatively different from the rule-sets used in this research, being combinations of system control parameter values (e.g. the base-level constant and the latency-factor). Each *overlay* is a set of numbers that are used by the equations throughout a run in place of the normal defaults, however they do not change the rules in the production system. Changes to rules-sets is the focus of my research. Both are complementary modifications, but my technique can be used to generate overlays as

[4] As of ACT-R 6.0 written in Common Lisp.

well, even though that is not its focus[5]. Interestingly, this combination of parameter overlays and rule-set modifications are suggested by Miwa and Simon (1993), but they never fully explored or implemented their ideas, making this the first work that could actively explore it.

Finally, ACT-R has some direct ties back to the Theory of Mind described in Section 2.7. Anderson (2009) even cites one of the specific versions of the Computational Theory of Mind, called the Multiple Drafts Model from Dennett (1992), as being supported by ACT-R. While Anderson was not attempting to implement any particular philosophical model, in the course of designing an architecture to meet certain diverse cognitive criteria[6] he came upon has historically been a successful design—which in retrospect closely matches Dennett's.

2.2.2 Modeling Individual Differences

The idea behind the modeling of individual differences is a natural specialization of wanting to model humans at all. Where much of the research going on now produce models that attempt to approximate mean human behavior in their outputs, modelers of individuals do not. Instead of aiming for the middle, models of individuals attempt to approximate arbitrary[7] specific individuals.

To illustrate this, consider the task of driving a car. Most modelers would model the average driver behaviors, which—presumably—are by and large fairly safe, and in keeping with prevailing laws. Such models would drive close to the speed limit, and would be rather predictable in its driving behaviors. These models would be unlikely to match anything but the statistically most common behaviors.

In the case of a model of an individual driver, that model would not only ignore average behavior in favor of whatever the original individual driver prefers, but it would also capture their variability. Such models could express realistic behaviors like speeding or driving aggressively. If the variability was correctly captured, the model may even shift its preferences in a way that is statistically similar to the original driver (e.g. being able to capture how their driving habits change after becoming exhausted or stressed). Given many models of different individuals, their average behavior should again even out towards the human norms.

While driving is an easy thought-experiment, it presents enough of a challenge that simpler tasks are more common. Simple problem solving and game play are more likely to be used as test problems in cognition research. With a properly designed experiment, it should be possible to use very simple tasks and still see individual differences in performance, choices, or other measurable behaviors. In the case of this work, a simple block sorting task is being used. Individuals there are expected to vary widely in what specific actions they take in order to sort blocks, but should be less variable in what method they personally use to sort blocks. So, even if the number or order of blocks changes, they are likely to sort them using the same rules of thumb[8].

Some historical research has been done on modeling individual differences. One of particular interest is the work by Miwa and Simon (1993), where the sample task is cryptarithmetic, a kind of math word-puzzle. In their work, they posit that the way to model an individual (as opposed to the human average) is to have some common core rule-set, a set of parameters specific to that individual, and a set of auxiliary rules specific to that individual. While they do not fully explore this idea in their paper, it does have a strong bearing on this research, where the idea of shared core behaviors being combined with an individual delta is key. The very idea of Strategy Groups is based on the idea that the shared core of the programs in the cluster represents a human problem-solving strategy.

2.2.3 Algorithmic Model Fitting

There have been a number of works related to this vein of research. The ones discussed below have an implicit common theme among them. They are all research that either examines how to fit models, or actually fits models to tasks related to the kinds of tasks that this research agenda aims to work with. Although none of them actually perform the same kind of fitting that this work proposes, they do provide context.

[5] Using a GP representation, the overlay parameters can be evolved as a separate set of genes from the production rules, within the same chromosome. Such work is beyond the immediate experiments I have planned, since Kase has already shown it possible with a less sophisticated approach.

[6] Interestingly, the founders of our field, Newell and Simon, had long advocated integrative modeling of cognitive processes (Newell, 1972, 1990) as being vitally important in any theory of cognition. Since the same mind must match all of the criteria necessary to solve all of the problems its known to solve, it is necessary that models minds must be constrained in their design by the need to solve all of these different kinds of problems.

[7] Arbitrary here means that the aim of such models is not specifically to copy a particular person, but rather to show that for any given person, it is possible to make a specialized model to represent them.

[8] There is support within this work for switching algorithms based on some internal criteria. Although there is no special mechanism to support this, the presence of conditional behaviors and self-perception in addition to the perception of the problems permit a uniform approach to representing this range of behavioral control–perhaps using normal conditional operators.

Among the example works that do fit models, the kinds of models include Paigetian block-tasks (Jones & Ritter, 1998; Jones, Ritter, & Wood, 2000), and debugging tasks (Langley, Ohlsson, & Sage, 1984). Both kinds of fittings attempt to match strategic problem-solving behaviors for their tasks, but they all only look at a very narrow range of behaviors and do not provide any general fitting methods. The block tasks cited here are actually more complicated than those this work intends to use, and involve special visual block-stacking behaviors. Since their focus is on developmental psychology, they use manual fitting of ACT-R control parameters to turn an existing model of an adult into one that matches child behaviors instead—with only manual edits to the production rules (involving breaking up rules or narrowing them).

As this research is in the modification of the production rules in a general way, there are some changes that are important. First, this work does not intend to use the visual system as extensively as they do, and will only use it to get accurate timings for GUI interface usage[9]. Secondly, though this work does not have an *a priori* model to modify at the outset, there will be an entire seed population of models after the bootstrapping phase. Third, this task is a simplified sorting task specifically because sorting algorithms are a known quantity, and they have been used before in cognitive psychology by Gascon (1976), Young (1976) and others. Finally, their changes are primarily to the ACT-R control parameters rather than focusing on the model edits. While the GP method fully supports parameter co-evolution with the models, enabling both should only be included in future work at the end of a successful completion of a normal model-only evolutionary run (e.g. solving The Block Sorting Task). That kind of co-evolution is planned as follow-up work.

As a matter of fact, there is a simple projection of models that could take the population results of the previous model-only runs and project them onto model and parameter ones, without loss of information. Conversely, there is a projection that can take a normal model edit only population and turn it into a parameter edit only population as well. Given this flexibility, its an easy enough thought-experiment to see how these prior works could be subsumed by this approach.

In addition to these works, there are several papers that provide a strong drive to do large-scale model fitting including Best and Gerhart (2011); Best (2012); Best, Fincham, Gluck, Gunzelmann, and Krusmark (2008); Best et al. (2009), as well as being hinted at in both Jones and Ritter (1998) and Jones et al. (2000). While these papers don't anticipate this work, they at least indicate that such things would be useful for practical as well as theoretical reasons.

Some of the more interesting work in this area are the papers by Best et al. (2008, 2009), which not only talk about why this kind of research is necessary, but also puts forward a method for doing certain kinds of model fitting and exploration. They introduce the idea of applying Adaptive Mesh Refinement (AMR) to the parameters of ACT-R as a way of sampling the space of possible parameter settings. They propose this because they identify (as mentioned several times in this work) that a combinatorial explosion occurs when exhaustive search is attempted—even in their much smaller task of only changing parameters instead of rules and parameters. AMR permits an efficient sampling of parameter space to find regions of interest to their work, by only looking at a representative subset of parameter space.

Where the other papers involving searching parameter space have been primarily manual in their sampling method, the only other one this author is aware of is the work by Kase (Ritter et al., 2017). The difference in their methods is in the underlying sampling algorithms and goals. Best's work with AMR is focuses on exploring parameter space because that space is interesting on its own, while Kase's Parallel GA work searched parameter space for special regions matching some optimality heuristic. Further, GA is a universal weak method, and as such can be adapted to any heuristic, while AMR only utilizes a landscape gradient heuristic to guide sampling (effectively ignoring all flat areas as boring). They are different but potentially complementary methods. In this work, which proposes a general method for handling search among both rules and parameters, it is possible to subsume AMR as a specialized hybrid operator for heuristically guiding parameter search which is complementary to the more general GP operators (e.g. mutation). Such an operator would use AMR's dynamic grid search to guide the search into interesting regions of parameter space.

2.3 Evolutionary Algorithms

One of the most important pieces of background information in this method is the idea behind Evolutionary Algorithms. Namely, the use of heuristically guided evolution as an universal weak method (a problems independent heuristic) for optimizing some function. There are a variety of Evolutionary Algorithms, ranging from the familiar Genetic

[9]This work is designed to work with potentially both motor actions, visual actions, and fMRI actions, but only motor actions are actually used in trace matching. The others are planned for later work.

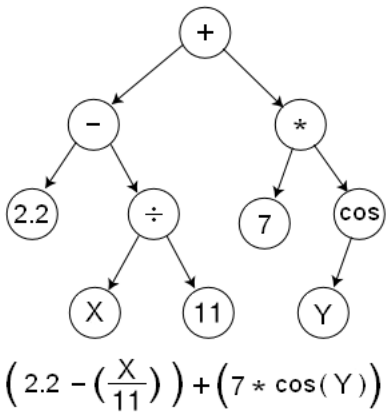

Figure 2.2: An Abstract Syntax Tree representing a Program (from Poli et al. (2008))

Algorithm, to more exotic methods. One of these more exotic Evolutionary Algorithms, of which there are a number that breed populations of programs to some end. These algorithms are applied in this work.

2.3.1 Genetic Programming

An Evolutionary Algorithm which evolves programs to optimize some fitness function, Genetic Programming is a very general mechanism for solving problems. Koza first described it as a generalization of Genetic Algorithms to evolve programs instead of fixed length chromosomes (Koza, 1989, 1992, 1995). Since its inception, dozens of variations have been developed (Poli, Langdon, & McPhee, 2008), including Grammatical Evolution from Ryan, Collins, and Neill (1998) and O'Neill (2001); O'Neill and Ryan (2001) which includes features of interest in this work, such as adherence to a BNF Grammar[10].

In Koza's original formulation, the GP process operated over Abstract Syntax Trees (AST) encoding computational expressions, like the one illustrated in Figure 2.2. These AST's were initially encoded in Common Lisp, and operated upon by Lisp Macros which modified the source code. Later versions support other languages and encodings. Regardless, the system would evaluate a AST and return a value, and then some Fitness Function would evaluate the result and assign it a Fitness Value.

Genetic Programming explores the Program Space of the problem, doing a heuristically guided walk through the space of all possible programs in the target language. The Fitness Function then assigns an additional dimension to Program Space, giving each possible program a Fitness Value. As the GP system explores more and more of the space, a Fitness Landscape will emerge, making a kind of topological map, where the mountains are all regions of high fitness, and the valley are all of low fitness[11].

This Fitness Landscape that lies inside of Program Space is a key part of my planned research, because it possesses a number of important properties:

1. Each point in Program Space is a fully realized program in the target language.
2. Areas of high fitness have the property that points adjacent to a point with high Fitness are more likely to also have high Fitness.
3. Adjacent points in Program Space are structurally similar, typically only a small edit-difference (e.g. one edit such as changing a number or flipping a bit).
4. If there are multiple kinds of programs which can solve a particular problem, the Fitness Landscape will be multi-modal, reflecting multiple solutions with multiple mountains (regardless of the optimal solution).
5. High Fitness regions in Program Space may feature lacunae, which are small regions of discontinuity in the fitness landscape. Thus, a point in Program Space with High Fitness may have several Low Fitness neighbors

[10] Grammatical correctness is a big help when generating AST's that will be evaluated by an outside system, like ACT-R, where the normal high-tolerance techniques Koza used don't naturally occur.

[11] There is nothing special about this arrangement, and its up to you whether you seek to maximize or minimize fitness. Mountains being good would be the maximized fitness case.

as well as several High Fitness neighbors. To use the mountain analogy, the mountain's surface is not smooth, and while this does not effect the average topography, it does mean you need to look out for crevasses while walking.

These properties are important in this research because the first three points are the basis for the ideas about Sampling, while the fourth predicts that multiple peaks imply multiple Strategy Groups, and the last point predicts one of the potential shortfalls Sampling may hit. Expanding upon this, without the important properties of Fitness Landscapes in Program Space, this line of research would not work out. Much prior work has been done on Fitness Landscapes giving reasonable certainty in their behavior.

In this work, the important thing to remember is that the proximity of two programs within Program Space implied structural similarity. Conversely, being a great distance away implies a structural dissimilarity of programs. Since similar programs are likely to behave similarly (though not guaranteed, see the point about lacunae above), there are likely to be clusters of similar programs with similar fitness and performance characteristics at solving the task. If there are clusters that are far apart, then the programs involved likely have found different ways of solving the same problem. These clusters are the Strategy Groups discussed in Section 3.1.

Further, the process of Sampling would be locating one of these Strategy Group regions in Program Space, and to randomly sample a point within the region. Since point five above warns of lacunae, the sample point can be fed into a local search algorithm to find a nearby higher Fitness point. Thus, we could reliably find high Fitness Programs that are structurally similar to the members of the Strategy Group. Evaluating it to produce a behavioral trace should produce something that can be tested to verify that it would be Clustered with the originally sampled Strategy Group. This would turn Sampling of Strategy Groups into a way of reliably making new members of the Strategy Group.

Additionally, should the Genetic Program locate Strategy Groups in the Fitness Landscape whose predicted behavior does not match the observed behavior of humans in the testing data, then it would predict that adding additional human data will eventually show the predicted Strategy Grouping in some subject's trace data. This makes the Fitness Landscape a strongly predictive feature in my work, and it is the quantification of the relationship between the problem being solved and the programs that can solve them, as mentioned in the introduction.

To better understand why GP is being applied in this problem, it is informative to look at an excerpt from Poli et al. (2008):

> The interrelationships among the relevant variables is unknown or poorly understood (or where it is suspected that the current understanding may possibly be wrong). One of the particular values of GP (and other evolutionary algorithms) is in exploring poorly understood domains. If the problem domain is well understood, there may well be analytical tools that will provide quality solutions without the uncertainty inherent in a stochastic search process such as GP. GP, on the other hand, has proved successful where the application is new or otherwise not well understood. It can help discover which variables and operations are important; provide novel solutions to individual problems; unveil unexpected relationships among variables; and, sometimes GP can discover new concepts that can then be applied in a wide variety of circumstances.
>
> Finding the size and shape of the ultimate solution is a major part of the problem. If the form of the solution is known, then alternative search mechanisms that work on fixed size representations (e.g., genetic algorithms) may be more efficient because they won't have to discover the size and shape of the solution.
>
> Significant amounts of test data are available in computer-readable form. GP (and most other machine learning and search techniques) benefit from having significant pools of test data. At a minimum there needs to be enough data to allow the system to learn the salient features, while leaving enough at the end to use for validation and over-fitting tests. It is also useful if the test data are as clean and accurate as possible. GP is capable of dealing gracefully with certain amounts of noise in the data (especially if steps are taken to reduce over-fitting), but cleaner data make the learning process easier for any system, GP included.
>
> There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions. In many domains of science and engineering, simulators and analysis tools have been constructed that allow one to evaluate the behavior and performance of complex artifacts such as aircraft, antennas, electronic circuits, control systems, optical systems, games, etc. These simulators contain enormous amounts of knowledge of the domain and have often required several years to create. These tools solve the so-called direct problem of working out the behavior of a solution or tentative solution to a problem, given the solution itself. However, the knowledge stored in such systems cannot be easily used to solve the inverse problem of designing an artifact from a set of functional or performance

requirements. A great advantage of GP is that it is able to connect to simulators and analysis tools and to "data-mine" the simulator to solve the inverse problem automatically. That is, the user need not specify (or know) much about the form of the eventual solution before starting.

Conventional mathematical analysis does not, or cannot, provide analytic solutions. If there is a good exact analytic solution, one probably wants to use it rather than spend the energy to evolve what is likely to be an approximate solution. That said, GP might still be a valuable option if the analytic solutions have undesirable properties (e.g., unacceptable run times for large instances), or are based on assumptions that don't apply in one's circumstances (e.g., noise-free data).

An approximate solution is acceptable (or is the only result that is ever likely to be obtained). Evolution in general, and GP in particular, is typically about being "good enough" rather than "the best". (A rabbit doesn't have to be the fastest animal in the world: it just has to be fast enough to escape that particular fox.) As a result, evolutionary algorithms tend to work best in domains where close approximations are both possible and acceptable.

Small improvements in performance are routinely measured (or easily measurable) and highly prized. Technological efforts tend to concentrate in areas of high economic importance. In these domains, the state of the art tends to be fairly advanced, and, so, it is difficult to improve over existing solutions. However, in these same domains small improvements can be extremely valuable. GP can sometimes discover small, but valuable, relationships.

The problem of figuring out what algorithm a person uses to solve a particular task fits well in these criteria.

- **The interrelationships among the relevant variables is unknown or poorly understood (or where it is suspected that the current understanding may possibly be wrong).**

  In a nutshell, this is the problem addressed by almost all models of human cognition. We have some rough ideas, and have been refining them for decades, but there is no simple analytic solution to model human minds.

- **Finding the size and shape of the ultimate solution is a major part of the problem.**

  While it is of interest to Cognitive Science, to know the range of ways that people can solve problems, this work is the first one to really provide tools to accomplish this, as well as a theory about how to interpret the results.

- **Significant amounts of test data are available in computer-readable form.**

  While it is not trivial to gather human data, it is commonly done, and there exist protocols to accomplish this.

- **There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions.**

  Cognitive Architectures like ACT-R and Soar are exactly this, and while they may provide different kinds of simulation data (ACT-R for instance provides precise timings, while Soar does not). Direct analytic solutions do not exist, and many simulationists would claim that they could not for many problems.

- **Conventional mathematical analysis does not, or cannot, provide analytic solutions.**

  If direct analytic solutions existed, we would undoubtedly be using them instead. In the case of the specific problem domain of figuring out what algorithms people are using to accomplish their tasks, then there are two ways that it has been done. First, the experimenter themselves can develop a method for solution for whatever narrow range of behaviors they can model. Second, the experimenter can take verbal self-reports from test subjects, and make models that represent them that way. Both options assume some prior knowledge of the solution, which this work does not.

- **Small improvements in performance are routinely measured (or easily measurable) and highly prized.**

  Commonly, there are many attempts made to capture human behaviors algorithmically. While the resulting algorithms tend to work reasonably well in some circumstances, there is inevitably some cases where there could be improvement. Such improvements have provided for entire industries to spring up to capitalize on them. For example, the simple act of answering a phone has spawned an entire industry dedicated to automated phone systems.

2.3.2 Grammar Constrained GP Variants

There are special considerations needed when trying to generate programs for my purposes. Since the general GP method can generate trees of arbitrary shape and content, it is possible that it could generate trees which would not qualify as Abstract Syntax Trees. If a tree would encode a program which would be disallowed under some grammar, then it is not an AST in the language which that grammar represents. Invalid programs cannot be run or evaluated, and tend to crash the system which generated them.

To prevent this, a number of attempts have been made to limit the generation of trees such that they can only produce grammatically correct trees. The range of options available is beyond the scope of this work, but two select methods have been selected for consideration in my method. These methods are Grammatical Evolution and Gene Expression Programming, and they are described in greater detail below.

Grammatical Evolution

This is defined as a linearly encoded version of Genetic Programming, that uses a special algorithm to transform a vector of numbers into a tree according to some specific grammar. This technique grew out of the work by Ryan et al. (1998) and O'Neill (2001); O'Neill and Ryan (2001) which includes proofs of grammar adhesion, increased neutrality of mutations, separation of genotype and phenotype, and other features of interest in this work. The key to their work is the utilization of a formal BNF Grammar[12], which is annotated to create a series of decision steps while exploring the grammar that result in a numeric vector expanding into a AST. This technique is mainly of interest as a exemplary method for constraining GP tree generation grammatically, with a minimal of added overhead.

Gene Expression Programming

This is defined as a competing method of generating grammatically correct program trees. GEP utilizes a very different set of principles from biology than most of GP systems. GEP also uses a numeric vector to generate ASTs, but its method for doing so is based on its own internal mechanisms that mirror biological machinery found inside cell nuclei. Similar useful properties like grammar adhesion, mutation neutrality, separation of genotype and phenotype, and others mark it as a viable alternative to Grammatical Evolution. It was developed by Ferreira (2001, 2006) in isolation, but has strong similarities with Grammatical Evolution. However, its specialized biological analog machinery permits it to strongly separate different genes during the operations of reproduction, such that they evolve without interfering structurally with one another[13], though they can (and typically do) interfere semantically with one another. Zhou, Xiao, Tirpak, and Nelson (2003) used this property to evolve rule-sets. Given that many of the programs I will be evolving will be IF-THEN rule sets, this prior work indicates that Gene Expression Programming may be a better candidate for my work than Grammatical Evolution, despite its increased overhead. The only reason not to use this method is the polymorphic type requirements for all operators.

2.3.3 Program Space

While Program Space has been discussed implicitly previously in this work, this section is intended to collect the key points in one place.

To reiterate, Program Space is defined as a a mathematical Space. Spaces in this sense include things like Euclidean n-Space, which is the Space of all n-tuples of Real numbers. By analogy, Program Space is the Space of all Programs in some formal language, as represented in Abstract Syntax Trees (AST). Since most languages allow the construction of statements of any finite length, the AST's in Program Space can be of any depth or dimension as permitted by the grammar of the language.

Within the context of Program Space, the fitness functions of Evolutionary Algorithms project an additional dimension onto Program Space called Fitness. This dimension consists of Real values between zero and infinity, plus the special value $\perp$ (Bottom) for non-terminating programs. The fitnesses of all elements from Program Space form a topology similar to an elevation map or a heat map.

One of the more unusual parts of Program Space is that it lacks any kind of absolute coordinate system, and so Programs can be compared spatially only through subjective measures, like Edit Distance between trees (Gustafson, Burke, & Krasnogor, 2005; Ekárt & Németh, 2000). Similarly, Program Space is intransitive, so that if tree A is 5 from B, and C is 3 from B, A is not necessarily 8 from C—as would happen if A, B, and C were on a number line.

[12] Grammatical correctness is a big help when generating AST's that will be evaluated by an outside system, like ACT-R, where the normal high-tolerance techniques Koza used can't be added.

[13] Grammatical Evolution for instance can cause cascading changes, such that a single mutation can result in a drastically different tree. To contrast, in Gene Expression Programming, a properly constructed chromosome will limit the structural effects from a mutation to the single gene it occurred in.

In addition to the most general version of Program Space defined above over trees, there are mappings of Program Space into Euclidean n-Space. The specific algorithm used in the EA may represent the AST in terms of a numeric vector. If Grammatical Evolution is used as an example, then the trees are represented as a vector of integers of finite length, which places a parametric bound of the depth of any tree. There are practical guidelines for selecting the length of that vector so that the programs will be sufficiently deep to represent whatever computation needs done.

2.3.4 GP Evolution of Expert Systems

For the purposes of this work, the differentiation between GP applications to Expert System Rule-sets and the application of GP techniques to similar rule-based systems is a matter of complexity. In general, the rule-bases utilized by Expert Systems are orders of magnitude more complex than those of simpler classifier systems. Indeed, classifier systems may even be subsumed into the Expert System rule-sets (e.g. as complex precondition testing). For contrast, it is not uncommon to find that a typical classifier system may contain a single expression tree, with less than 100 nodes, while a Rule-set for an Expert System would include at least ten to 100 times that many expression trees. This is to address a much wider range of behaviors than simply classifying an input (which might constitute a single rule in the rule-set). This difference in complexity is a key difference between the two kinds of systems, and techniques suitable for the multitude of rule-generation tasks may be computationally intractable in the case of Expert Systems[14]. Similarly, it is possible that Expert Systems may yield different heuristics than smaller kinds of problems, because of the domain knowledge that they may be able access during fitness evaluation, or similar resources that would not be available for narrower methods.

When examining the literature for full complexity rule-set evolution, this author has only been able to find two papers of note. They are discussed below, but that there are only these two papers published in this research area does not imply that there is no work worth doing in that area, only that the domain is still being explored. Based on what this author has read, it appears that computational techniques and resources have only recently become sufficiently powerful to be applied to large rule-set evolution[15].

It is best to discuss these papers as a complementary pair, with the work of Tsakonas et al. (2004) serving as the building blocks for the first publish full application to Expert Systems by Sickel and Hornegger (2010). Both are from Medical application areas[16] , and both are published using the creation of complex rule-sets for making decisions. The key difference between the two papers being that while Tsakonas uses the evolution of fuzzy rule-sets to solve classification problems for medical diagnostics, Sickel applies the technique more generally for the generation and modification of the rule-set for an Expert System which designs prosthetic hearing aids.

To generalize the work of Tsakonas for full sized Expert Systems use, Sickel introduced a number of new operators specialized for his domain. This includes gathering a number of existing selection techniques together, and creating a new selection technique they called *best-half selection*, which (although conceptually simple) filled a gap left by the other more sophisticated techniques they were combining. Additionally, they defined a domain specific form of fitness evaluation they called *dynamic fitness* which incorporates the Expert System in the fitness evaluation of their generated rule-sets. They also carefully selected from among their many available options for operators by running trials within their Expert System to determine which operators were most effective on their test cases. Their methods and analysis of results are natural extensions of the work on shorter rule bases in Tsakonas. Both papers saw measurable improvements in the systems they worked on, with Sickel's Expert System performance.

Another GP application related to this work is the work of Duffy and Engle-Warnick (1999), where Game Theoretic mixed strategy detection is accomplished using GP for Symbolic Regression—generation of complex algorithmic expressions, which are less complex than rule-sets. To differentiate his work from my own, he works with the Ultimatum Game, where the strategies are much simpler than the kind this work addresses. They are specifically expressible as a probabilistic choice between two specific options. This minimal complexity, as well as a lack of relation to human data differentiate his work from my own.

This line of research is the only published work to date directly applying GP to Expert Systems. As this work is in a vastly different domain, operates on a specialized Expert System (in the form of the ACT-R cognitive architecture), and explicitly works to mimic human data, it is safe to say that this proposed research is novel.

[14] It is well known that GP applications tend to be NP-Complete.

[15] Indeed, the current popularity of Deep Learning indicates that sufficient resources are only just recently becoming viable.

[16] Even though the Expert Systems discussed here are medically oriented, the domains that Expert Systems operate on may vary, but the underlying mechanisms are often very similar. Thus research on Expert Systems themselves is portable across domains.

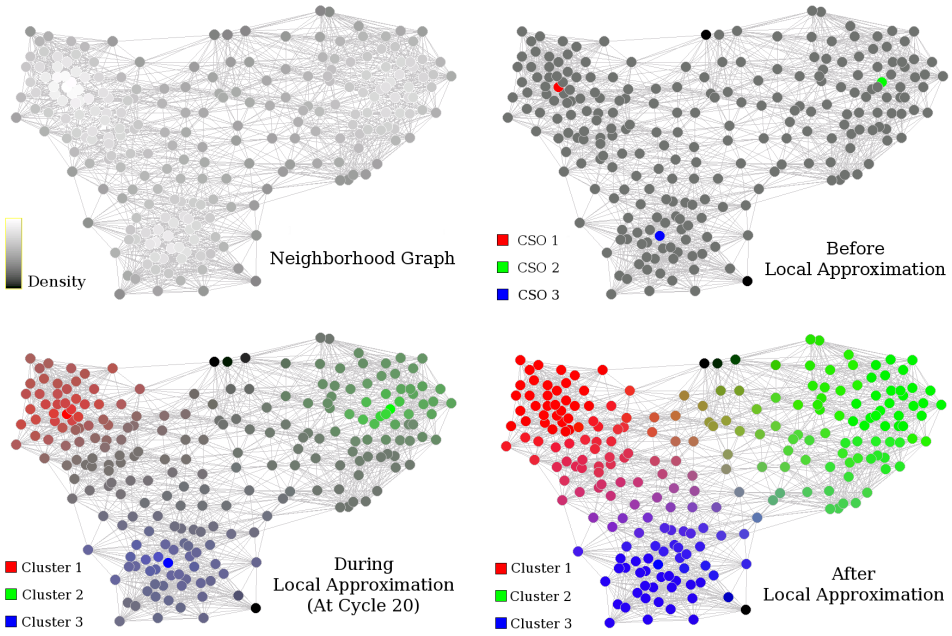

Figure 2.3: Demonstration of Fuzzy Clustering from (Fu & Medico, 2007).

2.4 Fuzzy Mathematics

Fuzzy Sets originate in the work of Zadeh (1965), where he describes them as sets for which there is no bivalent condition[17], and membership in a Fuzzy Set is defined by a Membership Function that maps items which may be in the set onto some scale, typically in $[0, 1]$. Built on top of this generalization of a set, Zadeh formulated Fuzzy Logic (2008) for which there is no Excluded Middle[18], algorithms for operating upon them (L. Zadeh, 1968), and mappings to *linguistic variables* (L. A. Zadeh, 1972), which are fuzzy concepts which are defined well enough to perform computations over them.

Others have since taken up Zadeh's Fuzzy mathematics and formulated interesting applications of them (Zimmermann, 2001; Siler & Buckley, 2004), including Fuzzy Clustering Methods (Yang, 1993; Fu & Medico, 2007; Baraldi & Blonda, 1999a, 1999b; Siler & Buckley, 2004; Xie & Beni, 1991). In Fuzzy Clustering Methods, the clusters are generated which may include fuzzy inputs as well as producing fuzzy sets to represent the groups. An example of what Fuzzy Clustering methods look like can be seen in Figure 2.3.

Anecdotally, Zadeh[19] identified Psychology and the study of the Mind as one of the domains of research that had the least uptake from Fuzzy Mathematics, as of his writing, and had expressed hope that this frontier would show itself a rich source of new knowledge in time. Now, Ross (2010) identified the same area as still being largely unexplored, about forty years later.

In the general sense, it is the kind of grouping that is normally accomplished by Unsupervised Learning methods like K-means (Lloyd., 1982) Clustering, or K-Nearest Neighbor (Cover & Hart, 1967; Beyer, Goldstein, Ramakrishnan, & Shaft, 1999).

Typically, Clustering algorithms make some kind of groupings out of unlabeled data within a reasonable computational time frame. Though such groupings are often unstable, where another run of the same algorithm on the same data-set may produce different clusters, it is known that there will be at least some vague kind of consistency when obvious groups exist.

There are two major kinds of clustering techniques for the sake of this discussion: crisp clustering methods, which utilize crisp sets to represent groups; and soft or fuzzy clustering methods, which utilize fuzzy sets to represent groups. Given the nature of the domain and the details of its application, fuzzy clustering methods are conceptually closer to the actual nature of the domain of Strategy Groups, which do not have conceptual exclusion, but only a degree of membership.

[17] The Bivalent Condition for sets is: an item is either a member of some set, or it is not.
[18] The Law of the Excluded Middle from formal logic states that any preposition is either true, or its negation is.
[19] Citation welcome, I cannot find his comment, but it was from either the late 60's or the 70's.

Fuzzy mathematics are all based around models of uncertainty, and have been reasonably successful in providing computational methods for dealing with it[20]. In this work, are identified a number of sources of uncertainty:

1. Uncertainty inherit in the human data, where people may (non-randomly) change their strategy while solving a problem, or where they may choose an admixture of multiple strategies. Additionally, nonstrategic behavior may occur (e.g. repeated no-op swapping in a sorting task) thereby introducing noise.

2. Uncertainty in distance or proximity measurement within Program Space, due to lacunae generating implicit discontinuities.

3. Uncertainty in the membership of a point within Program Space in a Strategy Group, since it may be members of multiple groups, or it could represent either a mixture of groups, or entirely new Strategies. For example, the Strategy of Comparison Sorts overlaps somewhat with Stable Sorts, but not with Radix Sorts, but Radix Sorts can be Stable Sorts.

4. Strategy Groups may be inherently fuzzy concepts similar to *linguistic variables*, like "cold water" and "hot water" or "old person" and "young person". These kinds of concepts are typically too ill-defined for non-fuzzy methods to perform meaningful computations using them, while fuzzy methods can perform useful computations despite this fact (L. A. Zadeh, 1975)..

This work anticipates using Fuzzy methods for dealing with these sources of uncertainty. There will be two kinds of clustering used in this work. First, the clustering of behavioral traces will be done on both real human data, as well as those traces generated by evolved programs. Trace clustering consists of taking a canonical representation of a sequence of states paired with the choices made at each state, and grouping them by some similarity criteria relevant to the task. A sorting task may group based on comparisons performed, or on swaps performed, and may collapse certain parts of the task into a single canonical form, like taking the fact that a particular set of things were compared to the current object, but ignoring the exact sequence of those comparisons.

The second kind of clustering is relatively task-independent, where this author would cluster the Abstract Syntax Trees of the Genetic Programming system based on structural similarity. Since many clustering methods work to minimize some kind of distance metric between the members of the data-set—such as edit-distance. The original edit-distance algorithm (Levenshtein, 1966) is still used in some spell-checkers as the basis of the Levenshtein Distance between strings, which measures the difference between strings based on how many edits it would take to turn one string into another. This edit-distance has been formulated in multiple ways for Trees (Philip, 2005) as well. Thus, this author anticipates the clustering to be done based on the minimization of the Tree-Edit Distance between two programs by the clustering algorithm.

2.5 Fuzzy Clusters

There are a number of clustering methods already out there, and some of the most popular fuzzy clustering methods are based on the same ideas behind their crisp counterparts. Typically these methods generalize the method that was originally defined over crisp sets to operate over fuzzy ones. Methods of interest here are:

- Fuzzy c-Means (Bezdek, 1980; Bezdek, Hathaway, Sabin, & Tucker, 1987; Bezdek, Ehrlich, et al., 1984; N. R. Pal & Bezdek, 1995), which is a fuzzy version of c-Means clustering.

- Possibilistic Fuzzy c-Means (N. Pal, Pal, Keller, & Bezdek, 2005), which is a Possibilistic[21] generalization of FCM.

- Fuzzy k-Nearest Neighbor (Keller, Gray, & Givens, 1985), which is a generalization of k-Nearest Neighbor clustering which groups items into fuzzy sets.

The diversity of clustering methods highlighted here is due to the special considerations of the method proposed. The Fuzzy Clustering is required because this work represents inherently fuzzy concepts, which may additionally contain (human or human-like) errors. This diversity is also because these methods do not work on the same kinds of problems; the c-Means derivatives are defined to operate over feature vectors, while the k-Nearest Neighbors derivative is defined over any data type that includes a distance measure between elements. Some of the data involved in this process is likely to be represented as a feature vector, but the really interesting things in Program Space lack such a

[20] Note that randomness is a particular kind of uncertainty, as so statistical methods are not always natural for modeling other varieties. Fuzzy maths should be seen as a general complement to Statistical methods, utilized when appropriate.

[21] Possibility theory is similar to Probability theory, but is based on different axioms, and is interpreted differently.

representation (see Section 2.3.3 for more details). Instead, there are alternative means of measuring the similarity of Programs (Baraldi & Blonda, 1999a, 1999b; Philip, 2005), though there is no obvious canonical method, but the best are derived from edit-distance measures. Many of the measures do provide some means of calculating the kind of distance information needed for k-Nearest Neighbors methods to operate.

2.6 Curse of Dimensionality

As the dimension of the data increases, the volume of the hypercube containing all data increases exponentially: r^d, where r is the range of the largest dimension, and d is the number of dimensions.

As a rule of thumb, most operations that analyze data become exponentially more computationally expensive as the dimension increases linearly (Bellman, 1961; Scott, 1992). This means that certain kinds of analysis becomes infeasible for certain kinds of high dimensional data. The techniques that experience this tend to require uniform sampling of the space, which increases quickly. An illustrative example: if each unit interval needed sampling, a regular cube of length 10 would require $10^3 = 1000$ samples, while a 10-dimensional hypercube would require $10^{10} = 10000000000$ samples[22].

While this certainly is a problem for techniques that perform uniform sampling, it is less of an issue for evolutionary algorithms exploring Program Space. Evolutionary algorithms explore the space randomly, to be sure, but will tend to explore only certain areas of interest preferentially. Thus, though they will be exploring larger spaces with larger dimensions, it is not normally the case that the size of the high-fitness areas is the same as the size of the total volume. It is problem dependent, but due to special considerations for this technique, such as not requiring the finding of a global optima or even finishing the entire problem run at once, merely getting things to explore reasonably interesting areas of the problem space, the Curse of Dimensionality does not appear to render this work infeasible.

2.7 Philosophy of Mind

In Philosophy of Mind, philosophers have put forth many theories and positions about the nature of the Human Mind[23]. Apropos philosophical background works include the Physical Symbol System Hypothesis (PSSH) of Newell and Simon (1976) and the Computational Theory of Mind (CTM) (Horst & Zalta, 2009).

The Physical Symbol System Hypothesis's main statement from Newell and Simon in their work from (1976):

> A physical symbol system has the *necessary and sufficient* means for general intelligent action.

In this context, a Physical Symbol System is a technical definition described earlier in the same work. It is basically any physical medium upon which a formal symbol system could be implemented. Example formal symbol systems include Boolean Algebra, Arithmetic, Chess, and the myriad of applications found implemented on computers. Physical media in this case could be a digital computer, or it could be rocks and sticks on a sandy beach. Taken together, it implies that a physical symbol system is any physical system capable of computation. Then, the *necessary and sufficient* clause implies that any generally intelligent system must implement it through computation, and that nothing other than the right kind of computation is required.

A related philosophic position is the Computational Theory of Mind (CTM). It was developed over many years by multiple philosophers, and there are many small variations of it. Their common core is defined here by the Stanford Encyclopedia of Philosophy (Horst & Zalta, 2009):

> Over the past thirty years, it is been common to hear the mind likened to a digital computer. This essay is concerned with a particular philosophical view that holds that the mind literally is a digital computer (in a specific sense of "computer" to be developed), and that *thought literally is a kind of computation.*

While it may sound similar (and indeed it is similar) to the PSSH, it developed out of a separate body of work and many of the differences lie in the definitions and explanations for varying parts of Mind. For instance, the Multiple Drafts Model (Dennett, 1992) puts forth an additional set of assertions about the nature of Consciousness assuming that the Mind is a computer in the sense of the more general CTM. His view is that what we call consciousness is not a privileged source of true knowledge, but that it is a subjective self-reflection. In other words, we perceive the

[22] When performing initial analysis, some back of the envelope calculations put the number of dimensions being explored by my technique between 20 and 200. For reference, there are only estimated to be 10^{80} atoms in the universe.

[23] Most of the literature is technical, and inaccessible to laymen. Fortunately, the definitions below summarize much larger bodies of work with only minimal jargon. Just do not mistake succinctness for lack of theoretical development.

current state of the various computational elements (e.g. storage buffers) at certain points in time, and stitch together a "stream of consciousness" to explain it to ourselves, even though it really isn't continuous[24].

The reason for including them here is because these theories are key to the interpretation of this planned work (see Section 5.3.3). If held to be true, the implications of this work should have a much broader appeal than the relatively narrow realm of Theoretical Cognitive Science. Since, if PSSH and CTM are true, then the programs it would be breeding would be automated approximations of the thoughts of real people, both in the sense of trying to solve the problems the same way as people, as well as in the sense of being approximations of the *test subjects* thoughts while solving the problems[25]. Psychologists, Anthropologists, Linguists, HCI Researchers, and anyone else looking to use test subjects that behave roughly like a human in some context would suddenly have an interest in this work.

It is not assumed that every reader will necessarily agree with PSSH and CTM, but if the relationships being explored in this work turn out to be strongly predictive, this work will provide some evidence towards empirical verification of both theories of Mind.

[24]Not unlike how an old film reel is just a series of still images that give the illusion of continuous motion.

[25]This sounds like some kind of *a posteriori* mind reading, and in a way it would be, if you trained it against a particular person's data exclusively.

Chapter 3

Methodology

3.1 Method

This section contains a detailed description that circumscribes the research agenda for this work, including information down to the individual algorithms. The main processes of the method are illustrated in Figure 3.1 on Page 34. Many of the representations and broad contextualization can be seen in Section 1.1 with the visualization of the values, types, and their interrelationships is in Figure 1.2. This chapter builds on the prior two, and readers may benefit from referring back to Figures 1.1 and 1.2 for the summary interrelationships between components, as well as Figures 2.1 and 2.2 for what ACT-R looks like, as well as what Abstract Syntax Trees (AST) look like, respectively.

3.1.1 Graphical Representation of Method

To understand Figure 3.1, it is important to understand the division between the Bootstrapping Tasks at the top of the diagram and the Post-Process Tasks at the bottom. In the method's processes the Bootstrapping Tasks need to be done initially, before any of the Post-Process Tasks can be utilized. However, some of the algorithms in the Bootstrapping Tasks section such as those involved in making clusters and producing new Programs via GP, can and will be run again after the initial bootstrap is completed. This sequence allows for these algorithms to be used to add to the dataset, and adjust the location and membership of clusters.

In Figure 3.1, the cloud labeled "Humans" represents the human test subjects who produce the Trace data. Regarding the symbology in Figure 3.1, all parallelograms are Data, all rectangles are algorithmic operators, all diamonds are IF-THEN-ELSE choice points, ovals are post-process applications, and the lone trapezoid is a multi-way choice. The Trace Matching EA is a GP running the Trace Matching Heuristics in Algorithm 2. Its output is an initial population of programs, which is checked for Strategy Groups in the diamond labeled *Detect Strategy Groups*. If the clusters fail a density check, they receive additional programs to add to the population via the *Problem Solving EA* which represents a GP running the Best Match Heuristic in Algorithm 3.

Should the Strategy Groups pass the density check, the population of programs and the Strategy Groups defined over them are ready. If no computational Post-Process tasks are to be done, the Strategy Groups themselves are an output for research use. They can be used as as described in Section 3.1.10, to describe and explain the range of heuristics used by humans to solve the original task (e.g. block sorting, in this work). Similarly, the Populations themselves can be selected from as stand-ins for human test subjects for cognitive testing involving that task (e.g. a model of a specific person driving a car could be used instead of that person when actually driving a car, perhaps allowing experiments that would otherwise be dangerous or stressful like simulating reactions during an accident).

When desired, the Post-Process Tasks can also be used to for further computational analysis, which generally results in a larger population and a revised set of Strategy Groups. The arrows pointing back to the Bootstrapping Tasks section represent that these operations can be repeated until either sufficient size or strength are recorded. Finally, Sampling can be used to create a new member of a Strategy Group, following the behavior distribution of the Group.

3.1.2 Bootstrapping Tasks

In order for this method to operate, a number of preliminary steps must be taken to generate an initial population of programs, as well as an initial set of Strategy Groups defined over that population. This part of the process is called Bootstrapping, and is intended to be done at least once. It can be repeated to increase the population size, or

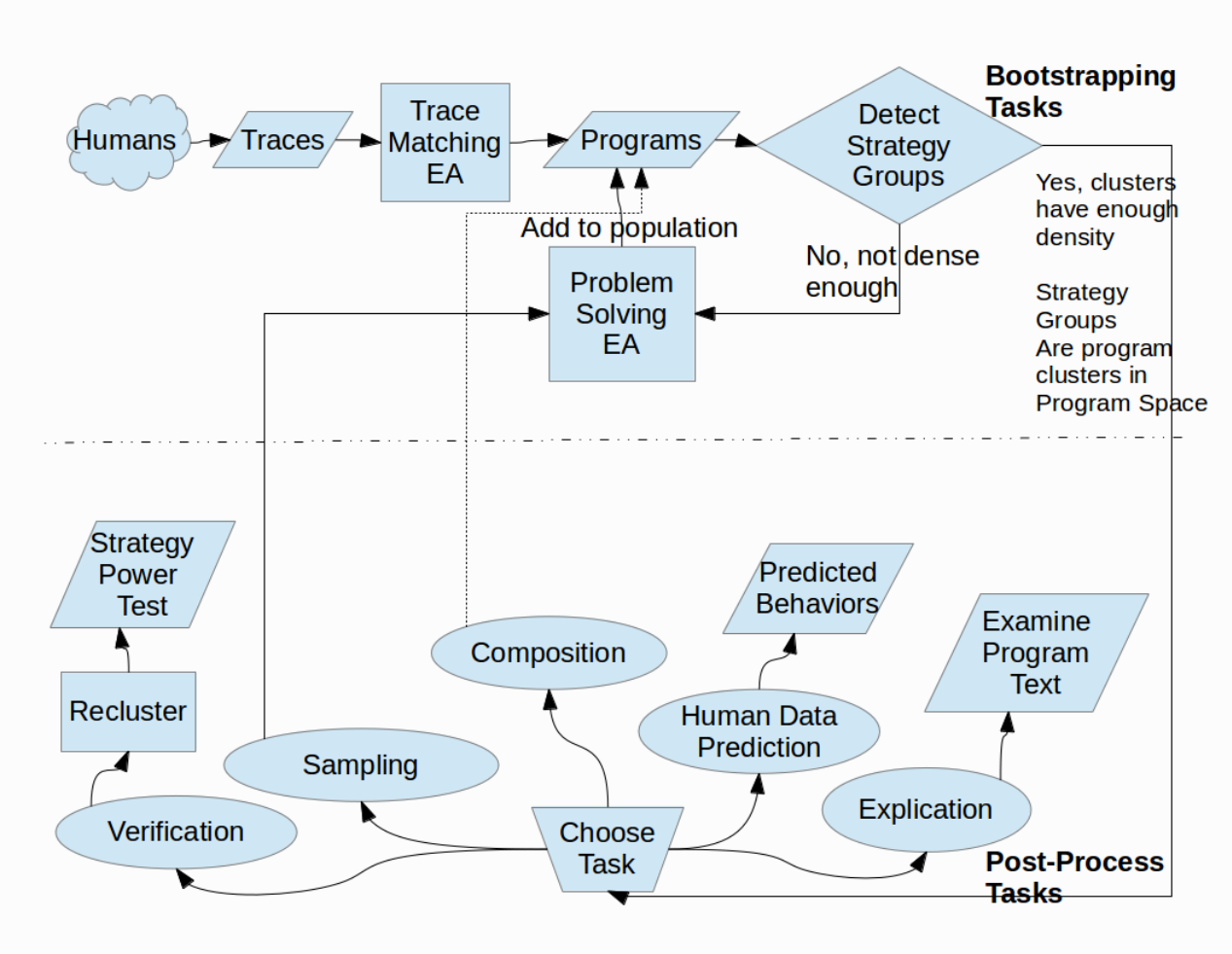

Figure 3.1: The Proposed Method. Bootstrap tasks are described in Section 3.1, and Post-Process Tasks are described in Section 3.1.5. Composition is described in Section 3.1.8.

to ensure the population is sufficiently diverse. If Strategy Groups are being used, their population can be likewise increased to increase their density. This includes Population Generation and Strategy Group Detection below, and is the upper half of the diagram in Figure 3.1.

3.1.3 Population Generation

Data: A set of Traces from a single individual, τ
Result: A set of Programs ψ, and a set of Strategies derived from those programs, ς
`// Use an EA to Maximize EditDistance (defined in Appendix)`
`// Some EA parameters omitted here for brevity (e.g. Population size)`
population ← `EAMaximize(`*EditDistance, τ, maxIterations*`)`;
`// N is a parameter`
elitePopulation ← `TakeBest(`*population, N*`)`;
s ← `FuzzyCluster(`*elitePopulation*`)`;
while *density of s too low* **do**
  `// Switch heurstic to best problem solutions rather than human matching`
  syntheticPop ← `EAMaximizeSeeded(`*BestMatch, maxIterations*`)`;
  eliteSynth ← `TakeBest(`*syntheticPop*`)`;
  elitePopulation ← eliteSynth ∪ elitePopulation;
  s ← `FuzzyCluster(`*elitePopulation*`)`
end
ψ ← elitePopulation;
ς ← s;

Algorithm 1: Bootstrapping

Take an individual and use them to generate several Traces, which are temporal logs of their activities. This data is then fed into an EA (currently GEVA, though future work may use other EAs as well) which then attempts to generate programs in a high-level domain specific language. The fitness criterion for these programs match the greatest subset of Trace data, as calculated by the longest correct prediction for each Trace, and the total number of Traces it matches. For some parameter that controls the number of iterations of the EA, the EA will run the full time only if there are less than some small elite size of programs with optimal matching such that they match all data.

A Trace is defined as follows, a list of 4-tuples (referenced in 3.1):

$$(initialstate, timestamp, operator, resultstate) \tag{3.1}$$

$$Trace :: [(initialstate, timestamp, operator, resultstate)] \tag{3.2}$$

$$Strategy :: fuzzyset\{Programs\} \tag{3.3}$$

$$Population :: \{Strategy\} \tag{3.4}$$

Once the EA halts, some number of the top (i.e. highest fitness) programs from the population will be extracted. At this point these programs represent the individuals observations as best modeled by the first round. They will then be combined with similar results from other individuals for the same test, and the resulting population of programs will represent a first approximation of the range of algorithms that the test subjects evidenced for the task.

3.1.4 Strategy Group Detection

This data will be run through an initial round of Unsupervised Learning[1] that results in Fuzzy Clusters. If no programs are sufficiently close together that they are *Very Close*, according to some standard Linguistic Hedge (L. A. Zadeh, 1975) measure which is judged to be meaningful (Beyer et al., 1999), then the initial population is judged to be too sparse to be informative.

So long as the clustering method fails to find sufficiently close programs, the population of programs is used to preseed the same kind of EA as previously. With the crucial difference between the EA's being the change of the

[1] Likely Fuzzy c-Means, with the AST edit distance being the distance measure for the algorithm.

fitness criteria from matching the Trace data (as described above) to correctly resolving the problem; this is because the seeded population already matches the human, and this step is to introduce some diversity. Reapplying the original fitness criteria is also an option, but mutation rates would need to be managed separately. This will terminate after some number of iterations, or when a sufficiently large number of programs achieve optimality. After this run, the resulting population is combined with previous program data to make an expanded population of programs. The Unsupervised Learning method will again be applied to check for closeness, and the iterations will continue until the hedge criteria is met. Conceptually, this is similar to making sure that there is enough data for a statistical hypothesis test to be significant. In this work that concept is just defined over Strategy Groups.

At this point, the Program Space has been explored to enough of an extent that at least some clusters exist within it. These clusters will be in terms of fuzzy membership functions, such that a particular program may belong to multiple clusters. This is intentional, and a feature designed to accommodate some form of modeling in the light of the fact that a test subject may exhibit both random noise as well as systematic variability. Furthermore, a subject may be using a number of methods interchangeably, with the result appearing as a mixture of them. Thus, these fuzzy clusters are the best way to admit all of this uncertainty within the model. Each cluster is an approximation of a single real algorithm, or a mixture of multiple algorithms, and it is referred to hereafter as a Strategy to distinguish it from any crisp form of algorithm. These Strategies are one of the main object of interest for this work, and are likely to be a concrete reification of automatically finding problem solving strategies per (Friedrich, 2008; Friedrich & Ritter, 2020), though that connection is a matter of future work.

3.1.5 Postprocess Tasks

All of these tasks are subtasks that require the initial generation of Strategies to be completed. Given their speculative nature, they are posed here as questions, and as well as an attempt to interpret the possible results.

3.1.6 Postprocess Verification

Does a member of a Strategy recluster with that Strategy as opposed to any other?

If a program is taken out of a Strategy Group and used to generate several behavioral traces similar to those that the original humans generated, do the programs that result from the reapplication of our process to these seed Traces wind up clustered with the same Strategy Group as the originating program?

Should the result of this test be positive, it will confirm the power of the method and indicate that it can reliably detect clusters in Program Space. However, if the result is negative, then the process would be too unreliable and would not have much power in the detection of Program Space clusters. If the positive version occurs, then the research would have met its first great challenge of power and reliability. If not, then it will need to be amended.

3.1.7 Postprocess Sampling

Does the EA process, if seeded only with members of a particular Strategy, and using either the best-heuristic or closest-heuristic or trace-heuristic, generate new members of that Strategy?

Similarly to the Verification process mentioned above, Sampling would generate new members of a Strategy Group. Using a similar process as mentioned, the main difference would be that the EA used to create them need not limit itself to trace fitting, but that it could use the other heuristics available[2]. After being generated, the new items should recluster with the original Strategy for this task to pass muster.

If this part fails, and the generated programs do not recluster, it may be the case that the program space has unusual or multimodal topology. So, it may be the case that Sampling may just not work, given some set of EA parameters (like maximum iterations or mutation rates). However, that kind of fault should be correctable using amended parameters or additional time. If multiple heuristics were being tested, this kind of fault is most likely to occur in those that do not feature any similarity criterion to the Strategy Group centroid.

If it succeeds, this process will allow us to treat the Strategy Groups as real objects of inquiry. Possibly, it would allow us to make new members of the group, but mainly it would give us the mechanisms to explore the nature and bounds of the group in a way that is not unlike exploring an unknown probability distribution.

3.1.8 Postprocess Durability Testing

Does the entire process work reliably over multiple runs, with the program space being explored being invariant? While the rest of this work treats it as axiomatic, there is still a need to confirm it. If it is, then this research will

[2]Such as fitness based on best behavior in the test, or based on proximity to the centroid of the Strategy Group.

have touched something fundamental.

If the topology of Program Space is static, then multiple runs across time and across distance should all be exploring the same mathematical structure. This means that researchers across the world would be able to run their own tests and compile the results into a combined monolithic data-set, and the results would be the same as if a single experiment had produced the data.

Program Space would be sensitive to only a small number of factors, in the ideal case. These factors are:

1. Cognitive Architecture: here it would be a stand in for mental factors that shape a person's behaviors, such as attention and memory.

2. High-level Representation Language: the most likely part to be a source of faults. The need to reduce the number of invalid programs generated during the EA phases of the process makes this necessary. However, the need to design the Domain-Specific Language and its compilation into the raw programs used by the Cognitive Architecture adds another layer of complexity that can introduce errors. The Language must be sufficient to represent the domain being addressed, but "sufficiency" can be difficult to determine. Furthermore, it is likely that there is no unique Language, so alternate encodings of the task would produce different Program Spaces.

3. The Task Itself: the nature of the problem determines what algorithms are sufficient to solve it. If the problem does not change, why should the algorithms used to solve it change?

These factors are the only ones that should determine the Program Space in the ideal case. If these are held constant by the experimenter, then the results should be exploring a single Program Space, regardless of the Random Seed used or the human data used to seed the process. This would be of interest to practical problems, such as how to drive a car, where the resulting exploration of Program Space would yield all of the possible ways to drive cars, given infinite time and space. More likely, it would permit researchers to use sufficient resources to determine the most reasonable ways that people drive cars, which could then be translated into control algorithms for automated cars.

3.1.9 Postprocess Reverse Prediction

When unseeded EA's generate populations using best-heuristic fitness function, rather than Trace-matching, do they predict human data? If they do not, then what makes the human-seeded populations special?

If the Program Space is static[3], and humans do explore it when solving the problem themselves, should then the EA not be able to operate without pre-seeding with human data? The process should be able to evolve programs to solve the task using a best-solution fitness measure instead of a greatest-trace measure. When not preseeding, the system will be unbiased and free to explore at random the Program Space. When the resulting population is compared to the greater body of results derived from seeding with human data, the non-seeded programs should do one of two things: either they will be similar to existing human seeded data, or they will not, and should predict that humans are capable of producing some previously unseen behaviors.

If this should be true, then the process could be run without human data at all[4]. Furthermore, it would be able to predict human data that has not yet been recorded. More than just being another proof of the power of the method, it would enable researchers to predict certain behaviors ahead of time, and test to see why they do or do not appear empirically. If it is false, however, this is even more interesting. Why should the human seeded data be privileged like this? It would open a whole new line of inquiry.

3.1.10 Postprocess Explication

Would Programs help explain how a person accomplishes a task? Would the Strategy Group (fuzzy membership) centroid reveal heuristics shared by members of that Strategy?

Each program from our population is a working algorithm that approximately solves the task. Since they are programs in a DSL or a Cognitive Architecture, the programs can be read like any other as plain text. At its basis , the text describes the steps taken to solve the problem, and thus provides some explanatory power automatically. However, the more interesting question is, does the population in Program Space in general, or the Strategy Groups in specific, provide more insight than a single program? The answer is almost certainly yes (i.e. using individuals to discover common algorithmic behaviors adds more potential insight than a single individual's results might), but requires empirical verification.

[3]The factors required for it to be dynamic require inconstant definition of the human architecture involved, or redefinition of the Problem, or changes to the Cognitive Architecture, or changes to the DSL.

[4]A large savings in time and funds, as well as a great way to reduce error.

It might be the case that the Strategy Groups have some structural properties[5] that represent a core heuristic explicitly used by the members of that Strategy Group. If so, then the heuristics could be extracted from the Strategy Groups and used directly. If not, then this can still provide some explanatory power in the descriptive properties mentioned earlier.

3.2 Grammar

Backus-Naur Form is from Backus (Backus, 1959) and Naur (Naur, 1963). It is a notation for Context-Free Grammars, and it is used here to express the grammar of a Domain-Specific Language for representing the mental and physical actions taken by human interacting with an instance of the Block Sorting Task, which this dissertation uses as an example problem (see Listing 1). In the general case, any specific problem that would be modeled in this method would require a similarly defined language.

3.2.1 Literal Representation

Note that this work is limited to strictly using BNF in this system by the semantics of the Grammatical Evolution algorithm (O'Neill & Ryan, 2001). It requires strictly the presence of a BNF grammar, and not something more powerful like Extended BNF (Wirth, 1977; ISO/IEC 14977:1996, 1996). This theoretical restriction results in grammars which must necessarily limit the range of possible literal inputs to those expressible through total enumeration, and not through more sophisticated methods. While it is possible to express unbounded numeric or string literals as the result of a recursive expansion of numerals or characters, the real performance results for doing so can cause unusually deep tree expansions, while also potentially raising the total complexity of the problem by orders of magnitude (e.g. by introducing an additional degree of freedom in the selection of each digit or character during the generation of a literal value).

For this reason, the BNF grammar presented here limits the number of available numeric or character literals to the smallest set required to express the range of our desired operations. Numeric literals include one to ten, so that the full range of slots/keystrokes can be represented, and zero and ten are included as outer bounds on those slots.

Similarly, the range of character literals was choosen to include only those characters which the Blocks-Sorting Problem strictly needed, and no more. While the declarative memory elements and the compiler both had support for the whole alphabet, this restriction was applied only to the range of valid input literals, not the internal mental processing that occurred afterwards[6]. Once a literal is input, all letters are reachable using iteration operators; numbers behave similarly, save that iteration is not total, and is restricted to the listed literals as well as zero. The syntax here is for string literals of length 1, rather than character literals. This type distinction between characters and strings of length one is the made because the underlying language treats them differently, and unifying all matching on string literals rather than mixed char/string literals is important for consistent behavior.

Boolean literals are total, with $true$ and $false$ being represented unsurprisingly[7]. Since booleans do not natively have any concept of ordering, they are not artificially made well-ordered just to behave consistently with the other data types. This is important in our formulation of the problem, because it means that the types being treated in ways that are unique to their natural properties is a key part of our grammatical design. While Grammatical Evolution is designed to handle this, other approaches (e.g. Gene-Expression Programming) require that all operators of equal arity accept any literal value, regardless of type, without error (i.e. they are polymorphic and total functions). Inducing such a total relationship undoubtedly changes the fundamental semantics of the problem being modeled. Furthermore, it requires a polymorphic formulation[8] of all operators which makes no distinction between operations which return useful values, and ones which do not (e.g. motor operations, control operators, error handlers). Polymorphic operators can similarly create an exponentially larger search space by removing the specificity of the grammar itself, as well as introducing a much wider potential for non-useful behaviors to arise by removing heuristic knowledge about the specificity of operators that a monomorphic system encodes via its type-system.

[5] Similarity in terms of edit-distance.

[6] To avoid confusion when looking at literals that are representable in this BNF grammar, please note that there are a number of non-grammatical literals that are used internally by the compiler and the programs it generates, which may be discussed along side these grammatical literals. They are not available as inputs, and are not legally representable according to this grammar. These include the literals: *:no-value*, *:empty*, and *:reload*. Internal operator symbols are treated similarly, and are distinct from input operators.

[7] In this case, the underlying system does not have literals for true and false, instead using `nil` or `'()` for false, and any other non-nil values as true. The $true$ and $false$ literals are actually just symbols without special meaning outside of the DSL.

[8] See 3.2.2 for a discussion of morphisms.

3.2.2 Morphism

Throughout this work, the concept of morphisms, such a monomorphism or polymorphism, refers the the terminology from Type Theory and Programming Languages Theory. While there are a number of other definitions available for other branches of mathematics and philosophy, this work uses them narrowly as technical terms. As such, they both refer to the properties of type signatures of functions.

Written in mathematical notation, a function is monomorphic if all types in its signature are bound and concrete, while a function with any unbound and parametric types is polymorphic. For example, the following are two examples of addition operators:

$$add_{mono} :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \text{ Is monomorphic over integers} \tag{3.5}$$

$$add_{poly} :: a \to a \to a \text{ Is polymorphic over type } a \tag{3.6}$$

In this notation, $\mathbb{Z}$ is the set of integers, and the type variable a could take on any value, such as integers ($\mathbb{Z}$), real numbers ($\mathbb{R}$), or complex numbers ($\mathbb{C}$). Each function takes two arguments (the first two symbols separated by a $\to$) and return some type of result (the last symbol after the last $\to$). Thus, the monomorphic addition, add_{mono} is strictly integer addition, and is not defined over any other kind of number. While the polymorphic addition, add_{poly}, can accept any kind of number, so long as both inputs are the same kind of number.

As mentioned in Section 3.2.1, this system is designed to be monomorphic. In addition to the performance reasons mentioned earlier, the choice of being monomorphic is further supported by the design decision to keep separate code-paths for ACT-R to learn, so that learned productions will only apply to a single type of argument. In doing so, this work arranges the order of operations in the calls to handle arguments to always be the same, thus not requiring extra steps to interpret types whose nature should already be fixed. This aspect is discussed further in later chapters.

All operations in this grammar are monomorphic, and there are only four base types: numbers, characters, booleans, and *IO*. The first three types have been discussed extensively in Section 3.2.1, but the last one bears further examination. First, the basic version of *IO* is unique in that it has only the single value of *Unit*. Unlike all other types in this grammar, there is no legal way to write a literal value of this type, because they can't be used for anything besides side-effects of their computation. A number of operators will return values of this type, but according to the grammar, those values cannot be used for anything. Thus, any operators which return the *IO* type will only be allowed to be legally called in places where their return values are discarded, such as in the prefix or suffix of the program.

In addition to that version of *IO*, there are several operators defined to be *IO-NUM* or *IO-CHAR*. These operators are more complex than a pure operator of the same kind, in that in addition to returning a normal number or character value, as they are also expected to do some kind of *IO* with the environment that the model is running in, such as looking at the screen, or pressing a button. They are also labeled as such when they are expected to interact with declarative memory more than a normal pure operator would. This grammatical distinction does not change the real type of the operators, and can be used monomorphically.

It should be noted that the symbol `:no-value` is included in code paths that are monomorphic over various types that don't include symbols within them, such as integers or booleans[9]. This is not actually so much a real value as a way to denote that no legal value of the real type was available, but that some kind of computation was completed.

From a type theoretical point of view, all of our types—integers $\mathbb{Z}$, strings[10] $\Sigma^\star$, booleans $\mathbb{B}$, and IO—are actually monomorphic over $\mathbb{Z}_\perp$, $\Sigma^\star_\perp$, $\mathbb{B}_\perp$, and $IO_\perp$ respectively. This can be interpreted such that $\perp$[11] is represented by the keyword symbol[12] `:no-value`, and all code-processing paths are designed to process it in addition their normal values. $IO_\perp$ is the simplest, since it has no literal representation, $\perp$ is its only legal return value in all cases. The other types all treat $\perp$ as an error in return values for functions, and this causes the monomorphic functions to be total.

3.2.3 BNF Definition

Now, referring to Listing 1, this section will explain how to read and interpret the actual grammar definition. Knowledge of BNF grammars is helpful but not necessary.

[9] Technically, it was chosen because ACT-R cannot match `nil` values, and so an alternative keyword symbol was chosen in the form of `:no-value`

[10] The Kleene closure of any length strings.

[11] This symbol is normally called "Bottom", and is the "Bottom Type" from "Type Theory". It's use here is as an explicit out-of-band value to indicate errors.

[12] Keyword symbols are a technical term from Common Lisp, which ACT-R and the DSL Compiler are both written in, where symbols need to be interred into a specific package. It is recommended practice to use symbols in the `keyword` package, so that `:no-value` is really syntactic sugar for `keyword:no-value`, so there is no magic happening, it is just a normal symbol in the `keyword` package.

```
<prog> ::= (swap <num-expr> <num-expr>)
<expr> ::= <io> | <num-expr> | <char-expr> | <bool-expr>
<base-expr> ::= <io> | <io-num> | <io-char> |
<io>   ::= (swap <num-expr> <num-expr>) |   (recenter-hands)
| (shift-hand <num-expr>) | (look-off-screen <num-expr>)
| (read-whole) | (once-only <expr>)
<num-expr> ::= <control-num-expr> | <io-num> | <var-num>
<control-num-expr> ::= (if-n <bool-expr> <num-expr> <num-expr>)
| (then-n <num-expr> <num-expr>)
| (once-per-problem-n <num-expr>)
<io-num> ::= (most-recent-index) | (noted-index)
| (index-of-letter <char-expr>) | (next-number <num-expr>)
| (prev-number <num-expr>) |  (scan-for-char-lr <char-expr>)
| (scan-for-char-rl <char-expr>) | (note-index <num-expr>)
| (look-at-char <char-expr>)
<var-num> ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
| (first-index) | (last-index) | (current-problem-length)
<char-expr> ::= <control-char-expr> | <io-char> | <var-char>
<control-char-expr> ::= (if-c <bool-expr> <char-expr> <char-expr>)
| (then-c <char-expr> <char-expr>) | (once-per-problem-c <char-expr>)
<io-char> ::= (most-recent-letter) | (noted-letter)
| (letter-of-index <num-expr>) | (next-letter <char-expr>)
| (prev-letter <char-expr>) |  (scan-for-num-lr <num-expr>)
| (scan-for-num-rl <num-expr>) | (note-letter <char-expr>)
| (look-at-num <num-expr>) | (next-letter-in-song <char-expr>)
| (prev-letter-in-song <char-expr>)
<var-char> ::= "\"a\"" | "\"b\"" | "\"c\"" | "\"d\"" | "\"e\""
| "\"f\"" | "\"g\"" | "\"h\"" | "\"i\"" | "\"j\""
| (first-letter) | (last-letter)
<bool-expr> ::= true | false | (and <bool-expr> <bool-expr>)
| (or  <bool-expr> <bool-expr>) | (not <bool-expr>)
| (xor <bool-expr> <bool-expr>) | (num\< <num-expr> <num-expr>)
| (char\<  <char-expr> <char-expr>) | (num\>  <num-expr> <num-expr>)
| (char\> <char-expr> <char-expr>) | (num= <num-expr> <num-expr>)
| (char= <char-expr> <char-expr>)
```

Listing 1: Block Sorting BNF Grammar

Beginning with the <prog> rule[13], which is our entry-point for the grammar, it can be read as follows: a <prog> value is defined (the "::=") as the sequence of expressions beginning with the symbols "(swap" then two <num-expr> value[14], followed by a ")" symbol. Each value with the pair of angle-brackets around them is a reference to another rule in the grammar. Symbols, numbers, booleans, and quoted characters are all literal values[15]. While all of the expressions which are contained in parenthesis are based on the S-Expression notation from Lisp (McCarthy, 1960). The elements in the parenthesis form a list, with the first element in the list (from left to right) is the operator, and the subsequent elements are its operands or arguments.

With that background, this first rule can be interpreted thus: a legal program starts with a swap of two numbers. Each of these number expressions could be simple or complex and nested. Because of how swapping must work, it cannot be defined over any other types besides <num-expr>, since it is tied to the design of the domain and represents striking numeric keys on the keyboard (that is, it cannot take letters as arguments and stay true to the experimental design). This BNF Grammar is used to maintain this type-safety by forbidding all illegal possible programs that could be randomly generated that would put something other than a number there.

While the first rule is mandatory in a BNF, the others are only relevant insofar as they are reachable from the initial entry point of the first rule. Thus, the second two rules are unreachable in the current incarnation of the grammar, but are kept in case the experimenter decided to permit actions to happen before or after the mandatory swap operation, in either a prefix (hypothetically occurring before the swap) or a suffix (likewise occurring after the swap). <expr> and <base-expr> will only get used whenever they somehow get reached from <prog>. The only directly reachable rule is <num-expr>, but it will be discussed later.

This rule, <base-expr>, is read similarly to the first one, but is contains the BNF-symbol | which means alternation, in this case, the rule may be read as a <base-expr> is exactly one of these options: <io>, <io-num>, <io-char>, or "". Each of those is a reference to another rule, save for the "" which is the empty string, which does nothing but terminate the grammar without adding text. Semantically, <base-expr> represents only operations which may occur in the prefix or suffix of the programs (if the base <prog> is amended to include a prefix and/or suffix), which never have a useful return value. They are allowed to be either some kind of *IO* operation, some other operation whose return value is ignored, or they can be empty to represent having no prefix or suffix, as the grammar currently shows.

The first alternation choice from <base-expr> is the rule <io>. It also features alternation, though its range of operations is wider, and includes both nullary operators, as well as operators of arity one or two. Most important among these is the `swap` operator, which takes a pair of <num-expr> arguments, which is consistent with its first appearance in <prog>. Its definition in this way allows multiple independent uses of `swap` operators per program, without permitting the useless case of nested `swap` operators. Adding a prefix or suffix is how this is accomplished.

Other operations which are available choices in <io> include nullary operators, such as `recenter-hands` and `read-whole`, which take no arguments, and will be evaluated during runtime to produce side-effects (e.g. `recenter-hands` includes a motor action without a useful return value). Please note, however, that they are not necessarily constant values, and they are strictly not literals. All other operators in <io> take either one or two arguments. Notable among these is the `once-only` operator, which is the only way to reach the <expr> rule mentioned earlier. It alone has this type because it has no return value, nor does it care about the type of its argument. All others require a stricter range of inputs.

Now that a basis for reading rules has been established, there are three categories of rules besides those already mentioned. Each of these categories has its own rules which are monomorphic on a single return type. These categories are: expr rules, control rules, io rules, and var rules. These labels are chosen because they are part of the BNF labels (e.g. <io-num> is an io rule, while <var-char> is a var rule).

The way that <expr> rules[16] work is fairly straightforward. Each argument is evaluated from left to right, and then the result is computed and returned. Control rules work slightly differently, since they may have unique changes to the control flow of the programs. For example, the `if-c` and `if-n` operators evaluate their first argument to figure out which of their two arguments need to be evaluated, and do not evaluate the other one at all, whereas the `once-per-problem-c` and `once-per-problem-n` operators will evaluate their argument once per problem and then just keep returning that cached result if reevaluated later on. These two operators also demonstrate the monomorphic design of the grammar, so that the operator for conditional numeric results is entirely distinct from the one for

[13]Note that some operators (e.g. <io>) are wrapped over multiple lines because they are little longer than other operators. Other operators with alternations may wrap as well, it has no bearing on the semantics to adjust the layout in this way.

[14]It is important to note for clarity that this system does not make any major distinction between a literal value of a given type, and a value of the same type arrived at as the result of some computation. Both are treated equivalently by the system in all cases, except in the internal details of the Compiler, which needs to issue code paths which either load a raw literal of a particular type or dispatch a jump to a subexpression known to return a value of that type.

[15]The exception being the one empty string "" (here represented as an empty alternation), which stands for the empty expansion, normally written ϵ in BNF notation. It means that when chosen, it terminates the selection process without adding anything.

[16]As well as var rules and io rules besides <io>.

characters.

Finally, whenever the BNF is being expanded, and it hits a terminal, such as a literal, a nullary operator, or the empty string, it halts expansion. When all rules have been expanded so that there are no unexpanded rules, then the BNF has generated a legal program in this language represented by the grammar.

3.2.4 Full Program Wrapper

Once a program has been generated using the BNF, there are a number of layers of wrappers that are added on top to make a complete executable program which gets compiled and run through ACT-R.

The first layer on top of the BNF is the addition of the following prefix:

```
(once-per-problem-c (read-whole))
```

The purpose of this layer is to add a mandatory behavior to every generated program without needing to directly interact with the generation of programs from the BNF grammar. Essentially, it adds one behavior that every person must minimally have in order to solve the problems: once per problem, read the current problem. It is minimal as it only occurs the first time a new problem is presented[17] and that it is the minimal number of visual actions in order to actually know what the current problem is. If anything, it probably overestimates the efficiency that a person presented with a problem would have, since the read-whole operator does not allow breaks.

The next layer is a wrapper at the Common Lisp level, where a macro named `with-dsl-wrapper` is wrapped around the whole input program. By doing so, the instance of Common Lisp is told that, when evaluated, the input text from the BNF and the modification mentioned above must all be handed the the Compiler for the Blocks-Sorting Grammar DSL. The details of its operations are detailed in Section 3.3, but it is mentioned here to ground the discussion. This layer is also responsible for placing the program from the BNF generation into a behavioral loop, thus allowing iterative problem solving behaviors without permitting generative loops in the grammar.

A final layer is added on top of the wrapper for the Compiler. This layer consists of several pieces of Common Lisp code which act as an instrumentation harness for the system. They permit values to be captured, measurements to be taken, inputs and outputs to be arranged, and configuration to be passed around the system. It is mainly there to act as glue to the GEVA library and the Clojure runtimes.

Most important to actual experimentation are the various cutoffs which heuristically guess that a non-productive run has occurred. Some of these are based on a maximum number of iterations, like so:

```
(defparameter *max-iterations* (let* ((n (length (car *problem-list*)))
  (n3 (* n n n))
  (k  3))
  (max n3 k)
  ))
```

which provides a iteration[18] cutoff of the higher of n^3 and 3. Other related timers are set to cutoff the run if a parametric amount of time has passed without any progress (n.b. time without progress is not the same as taking too many iterations, and catches hangs rather than thrashing).

These additional layers are described here for grounding and clarity, and (aside from the first layer mentioned that reads the problems) they do not change the semantics of the programs generated according to the grammar for the Blocks-Sorting Problem.

3.2.5 Operators Summary List

The operators described throughout Section 3.2.6 and detailed at length in Section 3.4 are numerous enough to benefit from a summary table. In Table 3.1, the information from Section 3.2.3 and Section 3.2.1 is condensed down into one place.

In addition to using Table 3.1 to separate out the operators by type and arity, there are a few functional groups that they can be gathered under:

Constants These operators and literals are either literal constants or nullary constant look ups that functionally act as terminals. These nullary operators may be universal constants or per-problem constants, which are constant for the lifetime of a single problem only.

1. “a” to “j”

[17] If a problem recurs, it still counts as *new* for our purposes.

[18] Again, the BNF body is evaluated within what is essentially a while loop, so there is no named iterator variable to draw on, just as person might not actively keep track of how many times they’ve stepped through their internal algorithm while solving a problem.

2. 0 to 10
3. true
4. false
5. first-index
6. last-index
7. current-problem-length
8. first-letter
9. last-letter

Ordinal Operators These provide successor and predecessor access.

1. next-number
2. prev-number
3. next-letter
4. prev-letter
5. next-letter-in-song
6. prev-letter-in-song

Logic Operators These are generally side-effect free comparison and Boolean logic operators.

1. not
2. and
3. or
4. xor
5. num=
6. char=
7. num<
8. char<
9. num>
10. char>

Notional Operators These operators are problem-duration note/recall pairs. The memoization operators function almost identically and are included here too.

1. note-index
2. noted-index
3. note-letter
4. noted-letter
5. once-per-problem-n
6. once-per-problem-c

Free Recall Operators These operators permit free recall based on ACT-R's memory equations biases and random recall.

1. most-recent-index
2. most-recent-letter

Visual Operators These operators mainly perform side-effecting visual and memory modifications.

1. scan-for-char-lr
2. scan-for-char-rl
3. look-at-char

4. scan-for-num-lr
5. scan-for-num-rl
6. look-at-num

Type Conversion Operators These operators are the primary way to convert via lookup what letter is in what slot and what slot is under each letter.

1. index-of-letter
2. letter-of-index

Control Flow Operators These operators are the only way to control program flow, and are integral to implementing algorithmic behaviors which work for multiple inputs instead of only a fixed constant input.

1. if-n
2. if-c
3. then-n
4. then-c

Manual-Only Operators These operators are available within the BNF and implemented in the Block Sorting DSL Compiler, but are excluded from automatic generation via the Grammatical Evolutionary (GE) processes. This is done purposefully, to reduce the degrees of freedom for genotype generation. These operators are manually added to the runtime by modifying the wrapper behavior, if desired. Swap is a special case, because it is not generatable, but is the root of all generated genotypes within the GE. Read-Whole could appear in the Visual Operators section if it were reachable via genotype expansion.

1. swap
2. read-whole
3. once-only
4. shift-hand
5. recenter-hands

3.2.6 Operator Selection

With the interpretation of the BNF grammar explained, the only part of the design which has not been mentioned is why these specific operators have been selected for inclusion. Generally, there is a trade-off between being able to model the full range of possible behaviors[19], versus presenting too many degrees of freedom and unnecessarily increasing the computational complexity of our evolutionary algorithm's search space. What has been included is what is presumed to be a minimal set of operators based on the range of potential behaviors that was observed experimentally, had been reported by subjects, supposed the potential existence of, or obtained from literature. This work makes no claims that this specific arrangement of operators is unique or optimal, and other formulations are undoubtedly possible. However, a BNF had to be chosen for this work to be done, and this is the one that was the result of that educated design[20].

For consistency with the previous discussion of this work will present this explanation of operator selection in the order they appear in the BNF grammar in Listing 1. Starting with `<prog>` and ending with `<bool-expr>`, the operators will be addressed in order, unless otherwise specifically mentioned. Operators which appear more than once, or which are type-specific instances of an already mentioned operator will likewise be skipped. All discussion of numeric, character, or boolean literals is in Section 3.2.1.

Beginning with `<prog>`, the first operator is `swap`. The inclusion of this operator was driven by the basic design of the Blocks-Sorting Problem, where the fundamental motor task is the pairwise swapping of blocks via keypresses. In order to be congruent with the actual experimental design, the activity of keypresses were constrained to the number

[19] More specifically, being able to model behaviors which may cause measurable changes to timings or motor actions. On-task operators will obviously create motor actions and memory activations. Off-task or behaviors that resemble a user *fidgeting* also needed representation since they were broadly observed over all subjects, and can cause quantitative changes to motor actions as well as potentially modifying the user's mental state while solving the problems.

[20] It is vitally important to report exactly which BNF grammar was used when reporting results derived from this method, due to the close coupling of the results with that choice. A seemingly small change may have cascading changes on the quantitative results.

Table 3.1: Table of Operators and their Types and Arities

Types	Literals	0-Arity	1-Arity	2-Arity	3-Arity
$\mathbb{Z}_\perp$	0 1 2 3 4 5 6 7 8 9 10	most-recent-index noted-index first-index last-index current-problem-length	once-per-problem-n index-of-letter next-number prev-number scan-for-char-lr scan-for-char-rl note-index look-at-char	then-n	if-n
$\Sigma^\star_\perp$	"a" "b" "c" "d" "e" "f" "g" "h" "i" "j"	most-recent-letter noted-letter first-letter last-letter	once-per-problem-c letter-of-index next-letter prev-letter scan-for-num-lr scan-for-num-rl note-letter look-at-num next-letter-in-song prev-letter-in-song	then-c	if-c
$\mathbb{B}_\perp$	true false		not	and or xor num< char< num> char> num= char=	
$IO_\perp$		recenter-hands read-whole	shift-hand look-off-screen once-only	swap	

keys on the top row of the QWERTY keyboard (i.e. $1, 2, \ldots, 9, 0$, but not the number pad). Because of this, the `swap` operator must take number values to perform its computation. Characters might have seemed convenient, but it simply was not the nature of the task being performed.

The next rule that has actual operators—as opposed to a BNF rule that is only alternation—is `<io>`, which includes `swap` again, in addition to all of the specialized IO only operators. The next operator in this rule is the nullary operator, `recenter-hands`. It is one of the few non-productive or *fidgeting* operators in the grammar. Unlike the other non-productive operators, this one is at least occasionally useful, since it more or less resets the position of the hands on the number keys. When not being used productively, that position reset can cause a person to have a noticeable delay on motor actions. In actual test subjects, this was one of the most commonly displayed motor actions, often taking place both after losing their place with their hands, as well as when *fidgeting* while staring at a problem.

The complementary operator to `recenter-hands` is the `shift-hand` operator. Following the logic from `swap`, it takes only numeric arguments, but unlike `swap`, `shift-hand` only ever moves one hand at a time. It will accept the number of a key, and then move the appropriate hand to dwell above the key without pressing it. It will not permit the incorrect hand to press the opposite hand's keys. Neither will it allow the hands to cross, or otherwise act oddly. This is all by design, because observation of subjects showed that they would often anticipate where they should be placing their hands, and would often just leave their hand over the most often used key, "1". Also, these observations did not show any evidence that they moved their hands in any particularly unexpected ways, nor did they cross.

The other *fidgeting* operator is `look-off-screen`, which takes a numeric argument of a number of seconds to look off screen. After it does this, it looks at a random location on the screen. This operator serves no productive purpose in this grammar, due to the unsophisticated proprioceptive assumptions that ACT-R makes about how a person knows what key their finger is on[21]. With real subjects, they sometimes display a form of *fidgeting*, but many times it accompanied motor actions such as recentering their hands, or shifting their hands to a new key. Thus, it is inferred that they lacked a perfect knowledge of where their hands and fingers were with relation to the keys. It has been included here mainly as a way to model the quantitative consequences of their visual movement. However, it specifically does not include mental processing for looking at keyboard key labels, nor at other features not specifically on the screen. This is at least consistent with the protocol instructions telling subjects to try not to look away from the screen (which they did often).

The visual operator `read-whole` is rather unique among operators, in that it does not strictly need to exist, since its behavior of reading the blocks from left to right could be accomplished in other ways using other visual operators. It is included for reasons of computational efficiency, since it is known that such an operation is strictly required to occur at some point in order to read a new problem. Rather than forcing the system to evolve the correct behaviors through trial and error, they are condensed into a single reusable operator.

The last of the operators in the `<io>` rule, `once-only` is also unique, in that it can process any kind of value in the grammar, including other members of `<io>`. All other operators are more restricted. This operator will act as a cache for a computation, by evaluating it normally when first called, and then never again—instead returning the cached result from the original computation. It is mainly there in order to capture the conceptual range of preparatory side-effects which happen just once.

After the operators of `<io>`, the next rule with operators is `<control-num-expr>`, which are all numeric control operators. All of these operators also have character counterparts[22], so what is said about these also applies to those. The first operator is `if-n`, the "if" operator that works with numbers. Its first argument is evaluated normally, and must produce a boolean value, but its second and third arguments are treated specially. If and only if the result of the first argument is true, then it will evaluate its second argument, and return that value. If and only if the result of the first argument is false, then it will evaluate and return its third argument. This special behavior prevents unconditional evaluation of arguments which the operator does not want to evaluate. As this operator is the basis of all conditional behavior in the grammar, its inclusion is obvious given the presence of the boolean type.

The `then-n` operator[23] is inspired by the `progn` operator[24] from Common Lisp. It provides a basic sequence operation, where the first argument is evaluated, then its value is thrown away, followed by the second argument being evaluated and returning normally. Its main reason for inclusion was to prevent our grammar from being unable

[21] To explain this statement, ACT-R models are just automatically aware of where their hands and fingers are with relation to the keyboard. This knowledge is encapsulated within the Motor Module, and is not automatically available via reflection to other Modules. The information may be requested via the Motor Buffer. However, the Motor Module itself may respond to motor requests without needing to refer to outside information about the state of the hands or fingers, such as by looking at them.

[22] Generally any operator with a "-c" or a "-n" deals with character or numbers, respectively. Thus, if you are looking for the character version of `if-n`, you should look at `if-c`.

[23] Note that this operator is not part of an "if-then" pair, its name should mean "and then".

[24] The Common Lisp `progn` operator is defined(*Common Lisp HyperSpec*, 2005) as the following: "progn evaluates forms, in the order in which they are given. The values of each form but the last are discarded."

to express complex multipart algorithms. While the first step is only done for the side-effects that its evaluation produces, that doesn't mean that it can't be used for preparatory steps. Additionally, they may be arbitrarily nested, permitting the full behavior of `progn` to be realized via stepwise nesting. Subjects also often described their actions as steps occurring sequentially, which this operator supports.

The last operator in the `<control-num-expr>` rule is `once-per-problem-n`, which is conceptually similar to the `once-only` operator, save that it only takes numeric arguments, and that it is "reset" whenever a problem is started. This can be used to store basic information relevant to the computation of the method of solving the problem. For example, the modeled person might be aware of the length of the problem, but also realize that the length of the problem does not change for the duration of a problem. The "reset" does not remove elements from declarative memory[25].

The next BNF rule is `<io-num>`, contains a number of operators which have character-based counterparts in the rule `<io-char>`. These character counterparts can be distinguished by their use of "letter" or "char" instead of "index" or "number". As well, several of these operators will take a character argument, and return a number, allowing for some measure of transferring and translating data between types. First in this rule is the operator `most-recent-index`. Like many other operators that follow, this one relies upon the memory activation in ACT-R to accomplish its task to remembering the most recently accessed memory of a number. ACT-R's activation gives an odds of recall biased towards recency, but which also has other factors involved. As a result, this operator, and others that rely upon ACT-R's memory recall mechanism are capable of returning erroneous results. Rather than being a limitation, this is actually a feature of ACT-R, permitting it to model human error rates. This operator, and others preserve this behavior with a nuanced model of error handling which distinguishes between different kinds of errors based on their similarity to humans.

There are a pair of rules `noted-index` and `note-index`, which try to recall a noted number, and commit a noted number to memory, respectively. The main item of note about these noted values is that they are not tied to a specific problem, and so it can be used to model some kind of persistent knowledge that can be carried across problem instances. After that pair, the `index-of-letter` operator interacts with what a person has seen, to recall the slot number of a input letter (e.g. "find out which slot I last saw "a" in"). All of these operators represent basic memory actions, based on the tasks at hand.

The next pair of operators, `next-number` and `prev-number`, may behave unintuitively compared to other memory-heavy operators, in that they model basic natural-number successors and predecessors using procedural knowledge instead of declarative. This design is justified by the assumption that, for the extremely narrow range of $\mathbb{N}_0 \leq 10$ (i.e. $\{0, 1, 2, \ldots, 10\}$), a person is arguably an expert at knowing what the next or previous number is for any given number[26]. These operators do not handle other numbers not explicitly listed on the `<var-num>` rule. This is justified as any broader accounting of numeric skills would be larger than the minimum functionality required to solve Block Sorting problems, and the assumption of expertise may not hold as solidly. In such a case, any more general of an implementation of numbers would require a whole theory of natural numbers, such as Peano Arithmetic(Peano, 1889) or von Neumann Ordinals(Von Neumann, 1923).

Our last set of paired operators for the`<io-num>` are `scan-for-char-lr` and `scan-for-char-rl`. These both perform an iterative visual scan from the edge of the letters, either from left-to-right (from the left edge) with the first operator, or the reverse with the second (from the right edge). The scan will halt either when it finds the letter which was given as its argument, or when it runs out of letters to check. If successful, the slot number for that letter is returned. Otherwise, the error handling is set up by the compiler so that the model can otherwise continue[27]. Very similarly, the `look-at-char` operator does a similar visual check for a letter from memory, looks at where it thought that letter was last at, and then returns the slot number of that letter. It does not scan, but it does handle failures similarly to the other two operators.

Now that `<io-num>` is finally complete, the next rule is `<var-num>`. As mentioned elsewhere, the literals have already been taken care of, and all that remain are the operators which represent simple value look-ups.

Following this reasoning, `first-index` and `last-index` both use information about the length of the problem, stored in the goal buffer. They return the lowest or highest slot numbers, respectively. This length information is also directly returned by `current-problem-length`.

While most of these operators are simply ways of getting parts of the sequence that the problem is represented by, the addition of perception of problem length and time add extra nuances that can potentially change a subjects behaviors. Problem length, for example, may be a reasonable estimator of problem complexity; a subject may choose to

[25] A code listing is available later in this chapter, but the short version of its behavior is that the "current" memory elements are used to match against, and so old problem's declarative memory elements are simply not recalled for the current code path, even though they had previously been current. Although it is not used here, it is possible to recall them manually.

[26] Based on a restriction on Peano Arithmetic to only include addition.

[27] The details of fault handling and error propagation are explained along with the compiler in Section 3.3.

switch strategies based on perceived difficulty, saving methods of least mental effort for simpler problems. They might also strategy switch based on a self-perception of *thrashing* behavior, where their current strategy is not resulting in useful changes to the current problem in a self-determined reasonable period of time. Finally, user complexity perception can be a way to measure non-productive cues, which represent boredom, stress, or frustration.

Moving on to the next rule that has not already been explained by analogy with the numeric operators, we come to the `<io-char>`. While almost all of operators these have been covered, there are four in particular which are notably different: `next-letter`, `prev-letter`, `next-letter-in-song`, and `prev-letter-in-song`. The first pair of these differs from their numeric analogs by using declarative lookups in order to find the ordering of letters by looking them up in a data structure called a Doubly-Linked List, which predicts uniformly linear lookup times. Contrast this against the operators, `next-letter-in-song` and `prev-letter-in-song`, which use the Tree[28] structure from Klahr (Klahr, Chase, & Lovelace, 1983). In this later case, lookups are non-uniform and non-linear in their lookup times. The details of comparing these representational operators will be discussed later, but the important part for this section is that they are based on different ways of learning the alphabet. For the first pair of operators, the alphabet is simply memorized as a sequence of twenty-six individual elements. For the second pair, the alphabet is learned via *The Alphabet Song*, wherein the learner learns the letters in groups, according to the structure of the song. Klahr (ibid.) treats the song at length, but the important thing about it is that it is commonly used to teach children the alphabet. Further, nearly all test subject mentioned using it during their sessions[29].

The last rule which bears discussion is the `<bool-expr>` rule. Boolean operators are the among the most restricted in where they are allowed to occur in the grammar, being legal only in the "if-" operators first argument. It may be surprising then, that these operators are relatively thorough and complete compared to those of other types. This is a design that resulted from the fact that the range of legal values in a boolean are both small and closed, so that operators are easily complete and total for booleans. None of these operators require additional memory access, and all operations are implemented via productions only. All of the basic operators, `and`, `or`, `not`, and `xor` are implemented by hand. All of the comparison operator, like `num<`, `num>`, and `num=` all delegate to ACT-R their actual comparison operation. These boolean operators and comparisons are vital to any kind of sorting problem to be solved, and so their inclusion is a necessity.

3.3 Compiler Contextualization

This section details the design and implementation details of the compiler which implements the Block Sorting Grammar and transforms it into programs for ACT-R to run. While this work attempts to balance between sharing too many details and too few, real working source code will be shown here as much as is possible, as opposed to abstract algorithms or formulae. The full source code will be made available upon request[30]. While much of this section's content requires a familiarity with compilers and ACT-R in particular, the more esoteric topics will receive at least explanatory footnotes. Finally, this section is vital to understanding the real implications for the design of general and reusable cognitive models which this compiler embodies.

While this work makes no claim that this specific compiler design is in any way special or uniquely optimal, it is the product of several rounds of revision. Indeed, the original design–an embedded compiler[31]–was completed years before this one, and worked reasonably well, save that it had two properties which were unacceptable for real use. First, since it placed most of the important computations outside of ACT-R, via delegation to the general computing infrastructure available through escaping to Common Lisp, the resulting productions were completely ineligible for ACT-R's production compilation mechanisms. This meant that no learning could occur, no proceduralization at all. In addition to making the system less realistic, such a restriction would also mean changes to the protocols for any experiments done with the technique, since any period where learning was thought to be occurring would need to be avoided. Second, the actual management of program flow, and for the most part, the majority of program state, would be handled via non-cognitive mechanisms and delegation to Common Lisp computations. For the accuracy and realism of timings and other quantitative parts of the model, this needed to be avoided.

[28] Though it is a tree, it is very similar to a specialized form of Skip List.

[29] Anecdotally, there were a couple instances of the song being sung aloud during the experiment.

[30] It is not included as a distinct appendix, due to the limitations of print space; when last printed, the source code was well over 700 pages long, comprising more than 27000 lines of code in the compiler alone, and well over 100000 in the project total, which includes modified FOSS.

[31] The word embedded here means that it was implemented by relying very strongly on runtime delegation to a host language, in this case Common Lisp. More technically, the Block-Storing Grammar was being run by an embedded interpreter where the only compilation was done by Common Lisp. Contrast this against the current compiler, where the compilation is largely done by my code, which doesn't directly interact at all with Common Lisp during model runtime, except in a very limited number of operators, where math or timing needs to be done.

```
(def binding-problem-list
  (let [result
        (clojure.string/join ""
          (list
           "(defparameter *problem-list* '("
           (clojure.string/join "" (doall (map #(str " \"" % "\" ") problem-sets)))
           "))")))]
    result))
```

Listing 2: Binding Problem-list

The present design of the compiler is specifically designed to avoid either of these issues, and so it uses ACT-R's cognitive mechanisms as much as possible, and flow control and state are all represented within ACT-R. For the most part, the compiler can be said to transform arbitrarily complex sequences of operators into a set of production rules and declarative memory elements that embody those operations. Since the Block Sorting Grammar was designed to utilize the same S-Expression notation as the Common Lisp language that it was written in, the parsing of the inputs is delegated to the Lisp Reader. Once read, the Lisp runtime evaluates the input according to a number of Lisp macro's[32], which transform the input structures into a list containing the instructions which will eventually be evaluated directly by ACT-R. At no point does the final model escape out to Common Lisp, outside of a small handful of uses of the *!bind!* operator to do arithmetic[33] or timings[34]. When these occur is limited to the infrastructure to load the next problem, as well as these specific operators: `note-problem`, `count-since-noted-problem`, and `look-off-screen`.

3.3.1 Inputs

As was mentioned in Section 3.2.4, the input to the compiler is slightly more complex than simply reading through the BNF Grammar would indicate. When an individual program is generated according to the grammar, it is rather bare, and is little more than a series of statements in the Block Sorting Grammar, and is not yet a legal Common Lisp program. It needs to be wrapped in a number of ways in order to legally compile.

The following string is a legal expression within the Block Sorting Grammar BNF: `"(swap 0 1)"`. Given this string encodes a legal Block Sorting DSL expression, then the way that it would be passed to the compiler is as follows: `(wrap-dsl-string "(swap 0 1)")`. Where this syntax calls the `wrap-dsl-string` function with the string that encodes `"(swap 0 1)"` as its argument.

First, the method of injecting data into the runtime is by binding that data to a dynamically[35] scoped variable, via the Common Lisp `defparameter` form. Since dynamic variables are a common way to provide default values in Common Lisp, they are used in the DSL Compiler for that purpose as well. Without having anything passed in externally by the Evolution context, the system is designed to perform self-tests.

To demonstrate this kind of binding, here is the one used for passing in the list of problems via binding the variable `*problem-list*`[36], in Listing 2. In this block of Clojure code (the application uses Java, Clojure, and Common Lisp code), the Clojure constant variable `binding-problem-list` is bound to the result of joining a list of strings consisting of the `"(defparameter *problem-list* '("` string that encodes the prefix of the Common Lisp wrapper, and the ending with the `"))"` string which encodes the terminals for the prefix. In the middle, the list of all problem sets is transformed from its datastructure representation into the string encoding of an escaped Common Lisp list of strings (e.g. `" \"DCAB\"  \"ABCDFE\" "`).

In keeping with this method for injecting values into the eventual program code, the `wrap-dsl-string` in Listing 3 defines a function that takes the input string, `str-input`, and concatenates the string to form a wrapper, as above, which will be bound to the Common Lisp variable `*input-ast*`. After wrapping, the string it concatenated with `binding-problem-list` from Listing 2 and a normally empty diagnostic monitor.

The most important part is the Common Lisp code[37]:

[32] These are compile-time functions, which take an AST as input, and return an AST as output.

[33] As mentioned earlier, avoiding this and replacing it with a cognitive model of arithmetic would require a complete an general model of arithmetic, which is outside of the scope of this work.

[34] This is due to needing access to a real clock, instead of the ACT-R metaprocess time, but a full ACT-R only replacement would require being able to task-switch the single temporal buffer among several task-specific timers. This is possible, but arguably outside of the scope of this work.

[35] Dynamic scoping is not the same as either global scoping, nor is is just lexical scoping. It is uncommon outside of Lisp or Perl, but the a typical example of its use would be temporarily globally rebinding the standard output stream, and the allowing it to revert to its prior values, down the call stack, until all dynamic bindings are removed, and it returned to its original value automatically.

[36] This is a legal variable in Common Lisp, and is a typographic convention commonly used to denote dynamically bound variables.

[37] Note the missing right parentheses, this is missing here, but matched in the string concatenation where the missing parenthesis is

```
(defn wrap-dsl-string [str-input]
  ;;; bind *input-ast* to the actual AST we got from the chromosome:
  ;; (defparameter *input-ast* "(with-dsl-wrapper (once-per-problem-c (read-whole)) ... ))"
   (let [bound-input
     (str " (defparameter *input-ast* '(with-dsl-wrapper (once-per-problem-c (read-whole)) "
          str-input
          " ) ) ")
     result
     (str bind-runtime-fitness-monitor
          binding-problem-list
          bound-input)]
   result
   ))
```

Listing 3: Wrap DSL String

```
'(with-dsl-wrapper (once-per-problem-c (read-whole))
```

This is the macro that denotes the boundary between Common Lisp code and the DSL language that the Block Sorting Grammar specifies. Everything within the body of the `with-dsl-wrapper` block encodes DSL expressions rather than Common Lisp ones. While the DSL is based on–and embedded within–Common Lisp, its implementation and compilation is entirely within the DSL Compiler instead of the Common Lisp compiler[38]. At the end of these steps, the resulting string is a legal Common Lisp program, and may be fed to the compiler. For example, if `str-input` was `"(swap 0 1)"`, then the resulting string would be[39]:

```
"(with-dsl-wrapper (once-per-problem-c (read-whole)) (swap 0 1))"
```

The `with-dsl-wrapper` part of the code instructs the Common Lisp compiler that the content of its argument list is to be processed by the `with-dsl-wrapper` macro provided by the DSL compiler to prevent Common Lisp from evaluating the DSL code as–likely illegal–Lisp code.

The expression within `once-per-problem-c` is the `read-whole` operator. Both of these have been passingly described in Section 3.2.6, but it is worth explaining the syntax in brief. As stated in Section 3.2.3, the syntax is called S-Expression notation which is used in Lisp (McCarthy, 1960). A legal expression in the syntax is either a plain constant, a naked variable, or a Form containing one or more expressions. While this can sound complex, the first two cases evaluate to themselves, and the last one is more or less Łukasiewicz's Polish Notation (1951) (n.b. not Reverse Polish Notation, which he made as well, but the operator-first version), just with parentheses to keep variadic operators unambiguous.

Unfortunately, this is only the beginning of the process needed to get usable results out of the programs of the DSL. For the purposes of integrating with the fitness evaluation aspect of the Evolutionary Algorithm which is actually generating the programs in the DSL, and then using the compiler to run them and return statistical performance information necessary for fitness evaluation. The following steps in Listing 4 are taken after arriving at around the use of Listing 3 from earlier:

1. The ACT-R *(clear-all)* command is added.
2. Several variables are passed in by binding global dynamic variables.
3. The human data is bound and passed in.
4. The problem-list is bound and passed in.
5. Break-times for the current subject are bound.
6. An error-handler wrapper is put in place in case of crashes.
7. The *injection-ast* is bound as a global dynamic variable.

appended later.

[38] As an embedded DSL, the host language is tightly coupled with the DSL's semantics, which is very beneficial. This means that the DSL need not reinvent the wheel about compilation semantics, it need only represent the semantics of its own DSL language in terms of the host language–Common Lisp.

[39] Now the missing right parenthesis is added, and this is legal code again.

```
(defn eval-dsl-string [& {:keys [interpreter dsl-input-string subject
                                 wrap-fn pre-fn post-fn
                                 load-prefix load-suffix
                                 injection
                                 per-problem-cutoff-millis
                                 total-time-cutoff-secs
                                 single-index-run?
                                 single-index
                                 random-wrap-fn
                                 ]
                          :or {wrap-fn wrap-dsl-string
                               pre-fn  identity
                               post-fn identity
                               load-prefix ""
                               load-suffix ""
                               subject ""
                               injection false
                               per-problem-cutoff-millis default-per-problem-cutoff-millis
                               total-time-cutoff-secs    default-total-time-cutoff-secs
                               single-index-run? false
                               single-index 0
                               random-wrap-fn wrap-dsl-string-random-access
                               }} ]

  (let [random-access-pair (when single-index-run?
                             (random-access-lbsp-data
                              single-index
                              (get human-data-map subject)))
        lisp-interpreter-input-string
   (str
    "(progn "
      (when testing-mode "(defparameter *testing-mode-only* t)")
      (if debug-mode
        "(defparameter *debug-mode* t)"
        "(defparameter *debug-mode* nil)")
      " (clear-all) "
      (str " (defparameter *per-problem-timer-cutoff* " per-problem-cutoff-millis " ) " )
      (str " (defparameter *run-time-seconds* " total-time-cutoff-secs " ) " )
      (if single-index-run?
        (trace-list-for-random random-access-pair)
        (trace-list subject))
      (when (contains? break-times-map subject) (break-times-list subject))
      (when catch-errors "(handler-case (progn " )
      ((comp post-fn
             (if single-index-run?
               #(random-wrap-fn % random-access-pair)
               wrap-fn)
             pre-fn)
       dsl-input-string)
      ;;; bind *input-ast* to the actual AST we got from the chromosome:
      ;; (defparameter *input-ast* "(with-dsl-wrapper (once-per-problem-c (read-whole)) ... ))"
      load-prefix ;; for example "#-"
      (when injection (str " (defparameter *injection-ast* (quote " injection " ) ) "))
      (if force-compile-compiler
        (str " (load (compile-file \"" dsl-loader "\")) ")
        (str " (load \"" dsl-loader "\") ")
        )
      load-suffix
      (when catch-errors " ) ")                          ; close handler-case progn block
      (when catch-errors
        (str
         "(error (some-error) (progn (format t \"ERROR: Handler caught:~a.~%\" some-error) "
         default-error-return-value
         " ) " ; close progn in error block
         " ) " ; close error block

         " ) " ; close handler-case
         ))
    ")" ; close outer progn block
    )
    interp     (if force-abcl-hard-reset-per-evaluation (new-abcl-instance! true)  interpreter)   ]
    (when (or debug-mode print-diagnostics)
```

```
      (println "DSL DEBUG; input=" dsl-input-string)
      (println "DSL DEBUG: lisp-interpreter-input-string=" lisp-interpreter-input-string)
      )
    (common-lisp-eval
     lisp-interpreter-input-string
     interp
     ))
  )
```

Listing 4: Eval DSL String

Each of these steps bears elaboration. First, the use of the `(clear-all)` is there to make sure that we aren't retaining anything from previous models[40]. Passing in data using global dynamic variables is needed because all ACT-R code is evaluated in the null lexical environment, so there is only the global scope to rely on. To ground this in Common Lisp terms, we defined and used this macro in Listing 5.

```
(defmacro supply-default (name value)
`(unless (boundp ',name) (defparameter ,name ,value)))
```

Listing 5: Supply Default

Thus, when the Clojure code implementing a fitness evaluator would add code like:

```
(defparameter *run-time-seconds* 120)
```

It would not be overridden by the variable defined in the compiler using:

```
(supply-default *run-time-seconds* 121)
```

Nor would the variable be unbound if the fitness evaluator did not provide a value, thus permitting a separation of concerns and more direct testing. The kinds of data passed in include information about time limits, the human trace data, the list of problems, the list of break-times, a raw and unadorned input program from the grammar (i.e. the output of `wrap-dsl-string`) , and finally the optional special string `*injection-ast*` which is evaled (if supplied) before the model is run via the ACT-R `run` command. This last one permits the injection of additional diagnostics into the runtime, though by default it is disabled, it is very useful for debugging, but it can be used for other modeling needs as well.

3.3.2 Outputs

The outputs of the compiler are one of two distinct values. First when the error-handler detects an error, the error code returned is the boxed integer value `java.lang.Integer.MAX_VALUE`. This will only happen when no useful evaluation occurs, due to any number of reasons (relevant here because randomly generating programs must be treated very defensively). If there is no error, then the returned value from the compiler is the Common Lisp list containing the following values, in this order:

1. *Success*: a boolean which is true if and only if, the system succeeded in solving all problems.
2. *Sdiff*: a real valued Levenstein Distance, as per Bard (2007). It treats the sequence of keypresses as a string, and compares the input human keypress sequence to that generated by our model.
3. *Ediff*: a real valued Euclidean Distance, with dimensionality promotion[41]. It compares the input human keypress timings sequence to that generated by our model.
4. *InAST*: the input AST as a list.
5. *Output-AST*: the output ACT-R model code, as a list.
6. *Total-Time*: wall-clock time, in $\mathbb{N}_0$ milliseconds.
7. *Correct-Problems-Before-Halt*: how many problems were successful before our time ran out, in $\mathbb{N}_0$.

[40] Technically, it should be redundant with the other isolation technique being used, where a fresh ABCL Interpreter instance is used per fitness evaluation, but it doesn't hurt to be safe, and it avoid the possibility of contamination via pre-compiled Lisp code being retained between Interpreter instances.

[41] Unequal dimensional inputs are given equal dimension.

8. *Problem-Count*: how many problems were input, in $\mathbb{N}_0$.
9. *Human-Trace-Length*: how many keystrokes were in the human trace, in $\mathbb{N}_0$.
10. *Trace-Length*: how many keystrokes were in the generated trace, in $\mathbb{N}_0$.
11. *Time-Portion*: additional metadata for debugging, in $\mathbb{R} \in [0, 1]$. The portion of the real human trace's runtime that was actually run.
12. *Bad-Swap-Count*: additional metadata for debugging, in $\mathbb{N}_0$. A tally of how many times the `swap` operator was given invalid inputs.
13. *Accumulated-Edit-Distance*: additional metadata for debugging, in $\mathbb{N}_0$. A tally that is reset to zero for each new problem. During a problem, it is a running accumulator summing the total edit distances for each time a swap occurs during that problem.

When looking at these outputs, most of them are fairly straightforward in their reason for inclusion as outputs, but a few bear explanation. *Sdiff* and *Ediff* are both fitness measures (see Section 3.5) that are easier to compute here than to force all of the work onto the fitness evaluator (no need to pass around large traces, just summary values). The return of the *InAST* and *Output-AST* are for inspection by researchers; they are not enabled by default because they can be large, and may not be directly useful. Most of the others are either used for diagnostic purposes or are optionally incorporated into the Fitness evaluation, as will be detailed in that section.

3.3.3 Memory Representation

When modeling memory cognition in ACT-R, one of the major points of contention is exactly how to represent things in declarative memory. The basic issue is how many different kinds of chunk (the named `type` of a memory element[42]) to use, as well as how many slots (named but typeless fields) each chunk should have. Choosing too many or two few could result in diminished functionality of the model, or incorrect quantitative predictions. As a rule of thumb, the number of slots should be limited to 7 ± 2 per Miller (1956); this is used only as a design heuristic, but memory design strongly influences algorithm design based on that memory layout.

When reading the code defining an ACT-R chunk, its important to understand that its based on the Lisp S-Expression syntax, where the first element of the list is the operator `chunk-type`, followed by a symbol[43] to be used as a label, and then one or more symbols used to label one slot each. For example, looking at Figure 6, the chunk type label is `letters`, while the slots are: `letter`, `slot-number`, `row-number`, and `done`.

```
(chunk-type letters
            letter
            slot-number
            row-number
            done )
```

Listing 6: Letter Chunk

To understand this chunk, its important to understand the nuance of how this representation connects to what it actually represents. Rather than being a general representation of character data, it is a more specific representation of seeing a letter in a slot in the experimental GUI interface. It notes the specific letter seen, as well as what slot it was seen in. Additional visual-processing specific data is included in the encoding of what row it occurred in, as well as if the visual processing was completed. Nothing outside of the visual processing code makes use of either of the two last slots. An important point to note about this chunk type is that it does not represent any kind of ordering outside of the implicit visual ordering implied by the slot number.

The representation of ordinality for letters is represented in two competing ways, with either the `alpha-order` chunk type from Figure 7, or the `alpha-song-chunk` from Figure 8. As introduced in Section 3.2.6, this experiment permits two different representations because there are two major ways to learn the alphabet: a memorized sequence of items as `alpha-order` does, or the Alphabet Song with a tree structure from Klahr et al. (1983). These two options

[42] Used here as the type that the ACT-R function `chunk-chunk-type` would return rather than that of the Common Lisp function `type-of`.

[43] A Common Lisp symbol is a unique value which supports only the equality operator. They look similar to strings, but don't normally include spaces or quotes.

have different performance characteristics, with the first having linear time lookup on most things, while the latter is non-linear (and typically smaller).

When looking at the alpha-order chunk, the first slot letter is an actual character stored as a string of length one. The next two slots, next and prev, are also characters like letter, but they are the actual next or previous letter[44]. If there is no legal value for either slot, they are filled with the symbol literal :no-value. This symbol is used internally by the compiler to annotate the absence of a value, because ACT-R productions will fail to match on slots with the value nil just as if those slots did not exist. Testing for out-of-band flag values is a common programming idiom. Finally, the ordinal slot is an integer starting at 1 denoting what ordinality that entry is. This slot is exclusively used for character equality and order testing, and is not used to cheat memory access.

```
(chunk-type alpha-order
            letter
            next
            prev
            ordinal )
```

Listing 7: Alpha-Order Chunk

Contrast this against the alpha-song-chunk chunk, where it models a linked tree structure. For clarity, the important slot is the type slot, which may have either of the symbols as :top or :pointer its values. All elements with :top represent entry-points into a subsequence of the Alphabet Song, while all elements with the value :pointer represent individual letters within a subsequence of the song. Thus, Klahr et al. (1983)'s song chunk γ[45] would have the type slot value of :top, but it would have no letter content itself, only a pointer to the beginning of the γ song subsequence (i.e. "l", "m", "n", "o", "p"), beginning with a element with a type of :pointer and a value of "l" and a pointer to "m".

Most of the other slots have values whose interpretation depends on whether they are a :top or :pointer element, the following ones don't: key-letter is a character string, and indicates the first letter in the song subsequence; next-song-chunk-key is as well, but it points to the next :top element; min and max are likewise the highest and lowest ordinality of the letters in the subsequence, and are only used when locating the nearest song subsequence for a given letter. All of the others are different depending on their type value. The start and end slots are booleans[46] indicating whether this is the first or last :top element (i.e. $\alpha, \beta, \gamma, \delta, \epsilon, \zeta$). When the element is instead a :pointer then the slots indicate the they are the first or last element in just that subsequence. Finally, target is a character string, and when the type is :top interpreted as a pointer to a :pointer element, otherwise it is a character literal.

As well, given the increased number of slots, it is acceptable given that the song-chunk-sequence is not used outside of assisting the clarity of a researcher reading the model. There are also three slots that could be removed by changing them into additional chunk types. They are included here to reduce the complexity of trying to interpret the memory of the models.

There is a detailed code excerpt in Listing 53, which contains all of the hard-coded implementation details of these memory elements. In the listing, they are used to represent the actual structures of the two different alphabet representations. Additionally that section also includes the staring metaproc (detailed below) that is used to bootstrap the system, as well as ,@dm-list which is where the insertion point is for the Compilers declarative memory elements list, which is dynamically generated from the input DSL program code by the Compiler.

```
(chunk-type alpha-song-chunk
            start
            end
            key-letter
            song-chunk-sequence
            next-song-chunk-key
            type
            min
            max
            target )
```

Listing 8: Alpha-Song Chunk

[44]These values are used similarly to pointers, and are not directly used to perform the prev or next operation bypassing memory access.

[45]I refer you to the diagrams within Klahr et al. (1983), since no permission is made by the publisher to reproduce them.

[46]Technically, due to the interaction of Common Lisp's boolean literals, "t" and "nil", with ACT-R, the boolean symbols being used are "true" and "false" instead of real booleans. This mainly matters in Production matching, where "false" may be matched, but "nil" cannot.

All of the chunk types seen up till now have been for the representation and storage of domain-specific information. The `op-sequence` and `metaproc` chunk types (see Figures 9 and 10) are markedly different, in that they represent and store information about the algorithmic behavior that the person is carrying out in order to solve the problem. In this way, a person modeled in this way is self-aware of the steps they are taking, and need to use declarative activity to remember what step they are on. Many ACT-R models do not work this way, as their flow of control in the program is hand-encoded by the modeler. Doing so restricts the model itself in a number of ways. They cannot reflect on their algorithmic behavior[47]. They cannot be adapted to follow arbitrary flows of control for the same domain, without explicitly writing them all manually. Finally, they do not need to access declarative memory for both the storage of domain data as well as processing data. The most accurate way to conceive of this change is that ACT-R models are often written as though reifying an optimal algorithm into hardware, while the methodology used by this compiler is analogous to a Stored-Program von Neumann Architecture (Von Neumann & Godfrey, 1993), with the role of the stored-program being played by these two chunk types.

Of these two types, `op-sequence` is the storage tape[48], while the `metaproc` is strictly working memory (i.e. registers in a von Neumann architecture). This storage tape is designed as a sequence of `op-sequence` elements that all kind of point forward toward the next step in their process. While its not literally the case (since their are many branching operations as well as internal loops), it is sufficiently similar to a stored program that the compiler may issue production rules which are generic with regards to their composition. They are composed entirely by the stored program, and modifying it will cause the flow of control to change as well.

Looking now at Figure 9, there a more slots than the heuristics guiding the design of memory elements would recommend. This is a little misleading, since the fields `arg3` to `arg6` are not used by any current operators (which all have arity at max three[49]). Besides that several of the slots are really one value broken down into multiple slots for technical convenience: `branch-name`, `branch-order`, and `op-name` form the first such value; `return-branch`, `return-state`, and `return-operator` form the second. The most important field is `op-name`, which is a symbol naming the operator being run. After that, `branch-order` is a $\mathbb{N}_0$, and is used to track what step is being done in the operator. Then the `branch-name` slot is another symbol which uniquely identifies a branch of the program. For clarity sake, when reading the output, `branch-order` values are created by appending the call-stack together, along with some additional uniqueness insuring operations involving Common Lisp gensyms. As a result, the values are often exceedingly long, but purposefully so. Together, these three values uniquely determine where in the program the system should be. While they are separated, they can be affixed together into a single very-long symbol if the working memory size heuristic is pressing. The other three slots, `return-branch`, `return-state`, and `return-operator` are identical to the other three, save that they denote a desired location to jump to, once computation is complete. Other slots in this chunk type include `done`, which is either of the symbols `t` or `:empty`, indicate whether or not certain internal processing steps are complete. Finally this chunk has seven slots with the names `arg0` to `arg6`. These are used to hold arguments to the operators. Depending on the context they are being used in, they can store literals, return values of previous child calls, or the symbol literals `:empty` or `:no-value`. Of these last two values, `:empty` indicates an unprocessed argument, while `:no-value` indicates a processed argument which had no useful return value, such as an IO operation or an error.

A final set of fields, `timestamp`, `last-argument`, `problem`, and `loop-iteration` are used to store the state of the program when new `op-sequence` elements are written to memory. Doing so allows otherwise identical parts of the code path to distinguish old data from new data based on whether or not the slots match the current state.

Now, looking at Figure 10, and go from looking at the stored program tape to looking at working memory directly. In this compiler, there is only ever one instance of this chunk type, though for other problems which require more extensive task switching, multiple could be used (as could multiple 'tapes'). A given instance is placed in the ACT-R Goal Buffer, and mutated in place without being garbage collected. All of the slots that it shares in common with `op-sequence` are more or less the same (save for these renamings: `branch-name` to `current-branch`, `op-name` to `operator`, `return-branch` to `next-branch`, `return-state` to `next-branch-number`, and `return-operator` to `next-operator`). The first four slots in `metaproc` are all metadata about the current problem or last problem. Slots `current-problem`, `starting-order`, and `last-problem` are all filled with Common Lisp strings, which encode the current problem's current order, the the current problem's original order, and the last problem's original order, respectively. The `length` slot is the $\mathbb{N}_{>0}$ length of the current problem. The next important slot is `subgoal` which,

[47] While it was not necessary for the solution of the Blocks-Sorting Problem, this reflection actually permits full modification of a model's algorithmic behavior during run time. All that needs to be done is the addition of extra `op-sequence` elements into memory. It can also be made to work with Reinforcement Learning by assigning utility ratings to different competing algorithm paths made in this way.

[48] Here, the term comes from Turing Machines (Turing, 1937), which are computationally equivalent to von Neumann Architectures, but the term is more suitable given the memory being allocated in something resembling a long chain or tape, rather than a contiguous memory block as in von Neumann.

[49] As a technical quibble, `arg3` and `arg4` are used in one or two places during the middle of a scan operation, but that is not a technical necessity, but rather to keep the implementation logic visible and organized.

along with dm-reload, is a symbol used to indicate positions within operators between memory accesses. These can take on a variety of values, mostly being :empty or :reload. The slot return-value is used to pass return values between operators, it is normally :empty, but after evaluations it must have some literal value or :no-value.

Finally, metaproc has a number of slots for keeping track of what are essentially different kinds of cutoff timers. Not all of these timers necessarily needed to be part of the chunk type, but they have been place there for technical reasons related to isolating use of the ACT-R !bind! operator to update timers. Of these, loop-iteration is the simplest, and is a simple counter which is reset when the a problem is solved. Its is associated with a cutoff timer which terminates non-productive behavior. The timestamp slot is a timestamp of the start of the current problem, and is primarily used when using "note-" operators. The last three are all all related to computing break times: time-since-process-start, time-since-last-break, and time-current.

The working memory size for metaproc is larger than op-sequence, but the justification remains largely the same. The only difference is inclusion of timer support, break support, and problem tracking. Most of these are included to avoid calling out to ACT-R !*bind*! operators (which precludes learning), and are not really used much outside of a couple operators. Of the slots not already discussed earlier in terms of op-sequence only three slots that are consistently used are subgoal, dm-reload, and return-value. These are absolutely vital to include in this chunk type.

```
(chunk-type op-sequence
            branch-name
            branch-order
            done
            return-branch
            return-state
            return-operator
            op-name
            arg0
            arg1
            arg2
            arg3
            arg4
            arg5
            arg6
            timestamp
            last-argument
            problem
            loop-iteration)
```

Listing 9: Op-Sequence Chunk

With these covered, the only remaining ones are the Instance Chunks (see Figures 11, 12, and 13), which are generally very simple. As they have a lot in common semantically, they represent the memory operation of noting a specific problem, number, or letter. Each of them includes a label slot which is unique identifier used to differentiate one instance from another. They all have a value slot for storing the noted value (though the equivalent slot on a problem-instance is called starting-order). As well, they all have a meta slot for storing metadata about that specific instance (the slot for this in problem-instance is cached-value). The problem-instance chunk type also has a slot for storing a time-stamp called instance. Finally, the three last slots, problem, iteration, and ticks, are designed to work with the similar slots in op-sequence, and permit differentiation of instances.

3.3.4 Algorithmic Task Control

In the design of the compiler, a balance had to be struck between expressiveness and control. If this work were to claim that the Block Sorting Grammar is capable of producing any reasonable algorithm for solving the problem, then it must not unnecessarily restrict the range of legal programs—potentially excluding the representation of entire categories of reasonable algorithms. On the other hand, all of the best practices for Evolutionary Algorithms indicate that restricting the search space and excluding unrestricted iteration is worthwhile. Unrestricted here denoting the EA being permitted to generate loops on its own. While the DSL could easily include looping operators, in addition to the iterative scanning operators it currently has, trying to determine if the iterations would even terminate is equivalent to solving the Halting Problem (Turing, 1937), and therefore formally undecidable, and thus only addressable using heuristic guidance. This is addressed below via a simpler, but occasionally incorrect deterministic heuristic.

While it has been explained previously in Section 2.3.3, the latter is newly introduced. Simply put, if a EA generates programs which may include unrestricted iteration, then those programs will never terminate for their

```
(chunk-type metaproc
            current-problem
            starting-order
            last-problem
            length
            current-branch
            branch-order
            subgoal
            next-branch
            next-branch-number
            next-operator
            return-value
            loop-iteration
            operator
            arg0
            arg1
            arg2
            arg3
            arg4
            arg5
            arg6
            dm-reload
            timestamp
            time-since-process-start
            time-since-last-break
            time-current )
```

Listing 10: Metaproc Chunk

```
(chunk-type problem-instance
            starting-order
            length
            instance
            label
            cached-value )
```

Listing 11: Problem-Instance Chunk

```
(chunk-type letter-instance
            value
            label
            meta
            problem
            iteration
            ticks )
```

Listing 12: Letter-Instance Chunk

```
(chunk-type number-instance
            value
            label
            meta
            problem
            iteration
            ticks )
```

Listing 13: Number-Instance Chunk

fitness evaluation to complete. Randomly generated programs are rarely useful, and it is trivially easy to cause infinite loops. How then can inherently iterative algorithms be represented? Utilizing restricted iteration is a reasoned compromise.

The methods of restriction used by the compiler are threefold. First, no operators feature arbitrary iteration, so no input from a randomly generated program can result in direct iteration; some operators feature guarded bounded internal iteration (e.g. `read-whole`, `scan-for-char-lr`, etc.). Secondly, the ACT-R runtime is instrumented to automatically halt after a set amount of time has passed; both total time and per problem timers are used. Lastly, a loop-counting iteration metaprocess is the main control structure of the compiler. It implements a wrapper around all of the instructions that need to be run, and runs them at most $max(n^3, 3)$ times[50], where n is the length of that problem. It may end early and load the next problem whenever the GUI plays a tone to indicate a successful problem completion. Break times are also handled by this mechanism. This mechanism is referred to hereafter as *main-loop*.

This *main-loop* is an excellent place to start examining the actual control mechanisms of the compiler[51] to see how ACT-R is made to implement a Stored Program computer. The first concept of importance is the idea of a Call Frame from computing, where they are a generic term for a variety of OS-specific form of Executable ABI. In our use, a Call Frame is a structure containing the following fields: a identifier for this Call Frame, normally a memory address in a computer; a list of our arguments, either literals or pointers (a memory address of another Call Frame); slots for storing return values from argument evaluations; local data and maybe our processing code (or a pointer to that code, depending on the OS); and finally a pointer to where we want to jump to when our evaluation is complete. For example, a computer implementation of the division operation may include two arguments for its operands, two slots for the actual numbers that get returned when those arguments are evaluated, a body of math code that performs that division, and a slot for a pointer to return to either its caller or (in the event of division by zero) the pointer to its error handler.

For this compiler, the `op-sequence` chunk type (Figure 9) is the basic representation of a Call Frame. On an actual OS like Linux or Windows, there are specific pieces of software that take Call Frames from long-term storage (e.g. RAM or Disk), and load it into specific parts of the CPU called registers.The CPU–by design–knows how to perform basic operations, move around memory elements, and perform jumps. When taken together, these operations are used to form a structure called the Call Stack, which is the progression of each Call Frame evaluating its arguments in order, recursing, loading another Call Frame, recursing, loading another Call Frame, and so on until the final terminal Call Frames are done. Each Frame returns its value to its parent Frame. Each Frame (usually) waits for all of its arguments to be evaluated and return their values before it evaluates itself and passes its return value to its parent too.

While there is not a one-to-one mapping, these roles are played by the compiler using the `metaproc` (Figure 10) and `op-sequence`. As was mentioned earlier, the memory model of ACT-R is different from a von Neumann machine, in that it lacks a concept of locality of data storage, and is also not capable of mutating data in place (besides in the Goal Buffer). Real Call Stacks exist in a very specific location in RAM, while the one created by the Compiler exists only implicitly–not quite heap allocation, because there is never any Garbage Collection, only statistical forgetting without actual erasure. While actually compiling, a real call stack is constructed to track all possible evaluation branches. It is used to construct both the Call Frame values stored as `op-sequence` declarative memory elements, as well as in the creation of the unique identifier for `branch-name` slots (and its kin).

This Call Frame data is represented in the declarative memory using `op-sequence` chunks as follows:

- Each operator gets one `op-sequence` chunk automatically, which also stores the data for the first argument, if there is one.
- For each argument beyond the first, there is one additional chunk with that arguments data.
- Each argument evaluation except the last argument for that operator adds one additional chunk with the stored result of its return value[52].
- Each operator gets one chunk with the information needed to return to its parent via a jump operation.

These separate items in declarative memory are loaded and unloaded via specialized handler routines in the *main-loop* and in the generic handler code the compiler provides for all operators. This code (i.e. `argument-p-` ⌋ `sequence-for`) will, in a type-sensitive way, take the value in the return slot of `metaproc` and and separately load from

[50]This cutoff is chosen because most sorting algorithms are more efficient than $O(n^3)$, so it is a bound which is usually larger than the runtime of a real sorting algorithm, but not so large as to waste time. The *max* part is to prevent immediate termination with unexpected input problems.

[51]Talking about the compiler in this way, I am referring to the general class of ACT-R programs which the compiler generates, not the actual source code that the compiler is written in.

[52]This "except the last" case is due to the fact that all others would get their return values overwritten when the next argument was evaluated. The last arguments return value is kept intact until the actual processing takes place.

declarative memory any `letters` chunk that has that kind of value in it (not booleans though, as there is no relevant value to recall). This behavior is a kind of priming or rehearsal of the activation for these related `letters` chunks. Once that is complete, the slots containing the information about where the original caller wanted the operation to return to is placed into `metaproc`, and the control again returns to the operator which had called for the evaluation in the first place.

Using this system, the complex behaviors permitted by a real Call Stack are translated into declarative memory retrievals, imaginal writes, and rehearsals of related values. It permits flexibility and arbitrary composition, without ignoring all of the good and proper memory operations which can have compounding quantitative effects on the results. Undoubtedly, this kind of behavior could be written more minimally by hand, and more optimal quantitative prediction could be possible, but many models like that are neither generic or flexible enough for any kind of reuse or composition. For a more nuanced comparison, it might be more apt to consider models of multitasking or threaded behaviors. The burden of context switching is then less obviously out of place compared to these models. Similarly, few models have the capacity to reflect on their operations, and potentially change them. This design accomplishes both by taking core ideas from computers and translating them into something cognitively plausible, if complex. However, most problems need a certain minimum amount of complexity in order to solve them. The implications of this design on the learning process of ACT-R is discussed elsewhere in Section 3.3.6.

3.3.5 Compiler Internal Representation

The entry point for a technical discussion of operator implementation is to understand roughly what the compiler is actually doing to define an operator, and then look at the implementation details of a few operator specimens[53]. The whole compiler uses just three kinds of records to operate: *dsl-compiler-state*, *dsl-op-sequence*, and *dsl-operator*.

As the first of these record types, as shown in Listing 14, a *dsl-compiler-state* stores a symbol table, a set of used identifiers, a stack representing the current identifier, and two lists which contain the input AST and the output AST. While these last two are used mainly for input and output to the compiler, the other fields are used to track the current call stack, as well as maintain the information needed to generate new unique identifiers as well as to test if an identifier already exists.

```
(defclass dsl-compiler-state ()
  ((symbol-table
    :initform (make-hash-table :test 'equal)
    :type      hash-table
    :accessor  symbol-table)
   (input-ast
    :initform nil
    :type      list
    :accessor  input-ast)
   (output-ast
    :initform nil
    :type      list
    :accessor  output-ast)
   (current-ident
    :initform nil
    :type      list
    :accessor  current-ident)
   (used-idents
    :initform (make-hash-table :test 'equal)
    :type      hash-table
    :accessor  used-idents)
   (branch-stack
    :initform nil
    :type     list
    :accessor branch-stack)
   (dm-items
    :initform nil
    :type     list
    :accessor dm-items)
   ))
```

Listing 14: Class Definition for DSL Compiler State

[53] I fully intend to avoid repeating details mentioned elsewhere, as well as not to inundate the reader with low-level technical details. Those details shared are chosen because they are sufficiently high-level to be useful in conveying broad designs.

```
(defclass dsl-op-sequence ()
  ((branch-name
    :initform :empty
    :initarg :branch-name
    :accessor branch-name)
   (branch-order
    :initform :empty
    :initarg :branch-order
    :accessor branch-order)
   (done?
    :initform :no
    :initarg :done?
    :accessor done?)
   (name
    :initform :empty
    :initarg :name
    :accessor name)
   (arity
    :initform 0
    :initarg :arity
    :accessor arity)
   (args
    :initform nil
    :initarg :args
    :accessor args)
   (args-literal-value
    :initform nil
    :accessor args-literal-value
    :initarg  :args-literal-value)
   (return-branch
    :initform :empty
    :initarg :return-branch
    :accessor return-branch)
   (return-state
    :initform 0
    :initarg :return-state
    :accessor return-state)
   (return-operator
    :initform :empty
    :initarg  :return-operator
    :accessor return-operator)
   (parent
    :initform nil
    :initarg  :parent
    :accessor parent)
   (parent-arg-number
    :initform 0
    :initarg  :parent-argument-number
    :accessor parent-argument-number)
   (dm-identity-override
    :initform nil
    :initarg :dm-identity-override
    :accessor dm-identity-override)))
```

Listing 15: Class Definition for DSL Op-Sequence

Meanwhile, the *dsl-op-sequence*, in Listing 15 is a very straightforward mapping of the *op-sequence* chunk type from Listing 9. In addition to the fields there, the record also keeps track of the arity of the parent operator, as well as a flag noting whether the value is a literal value, or would be compiled as a jump to a subexpression. There is also an additional field allowing for explicit handling of some edge-cases in naming identifiers.

```
(defclass dsl-operator ()
  ((name
    :initform nil
    :initarg  :name
    :accessor   name)
   (arity
    :initform 0
    :initarg :arity
    :accessor   arity)
   (dm-fn
    ;;This is a function:
    ;;(operator, operator-branch-name,
    ;; operator-branch-state,
    ;; , args) -> LIST OF DM-OP-SEQUENCE objects
    :initform #'identity
    :initarg :compiler-for
    :accessor   compiler-for
    )
   (productions
    :initform nil
    :initarg  :productions
    :accessor   productions
    )
   (jump-locations
    :initform (make-hash-table :test 'equal) ;; type is symbol -> int
    :initarg  :jump-location
    :accessor   jump-locations
    )
   )
  )
```

Listing 16: Class Definition for DSL-Operator

Lastly, the *dsl-operator*, in Listing 16, record has a very limited number of fields: an operator name, the arity, a list of production rules, a table of jump locations, and a generator function for making *dsl-op-sequence* records specific to this operator called $dm\text{-}fn$; its type is as follows:

$$dm\text{-}fn :: compiler\text{-}state \rightarrow args \rightarrow parent\text{-}sym \rightarrow \tag{3.7}$$

$$my\text{-}sym \rightarrow return\text{-}sym \rightarrow return\text{-}state \rightarrow \tag{3.8}$$

$$return\text{-}op \rightarrow parent \rightarrow parent\text{-}arg\text{-}number \tag{3.9}$$

$$\rightarrow [dsl\text{-}op\text{-}sequence] \tag{3.10}$$

To explain that, $dm\text{-}fn$ is the function that ultimately creates the Stored Program part of the von Neumann Architecture we are implementing, and so it requires complete knowledge of the Call Stack to work. In order:

- A *dsl-compiler-state* record representing the current state of the compiler.
- A list of arguments as strings.
- A unique identifier for this operator's parent.
- A unique identifier for this operator.
- A unique identifier for this operator's return site.
- A identifier for a return-state.
- A identifier for a return operator.
- A identifier for the parent's operator name.

- A number denoting which argument of the parents this operator is.

It uses this information to fully walk all possible code paths, without actually executing any operators. While doing so, it maintains the Call Stack, and uses it to generate unique ids for all elements being generated. The actual individual possible code paths are encoded in *dsl-operator* chunk type elements, but the runtime behavior of the executed productions determines the execution path by determining during run time which elements will be recalled from memory. Not all elements are always used during runtime as a result.

```
(defmacro define-operator (&key
                           (name "UNKNOWN-OPERATOR")
                           (arity 0)
                           (prod-jumps nil)
                           (dm-jumps nil)
                           (compiler-for nil)
                           (productions nil))
  "This is a utility macro to automate the commit order of modifications to the operator defs. Don't try to do it
  ↪  manually."
  (alexandria:with-gensyms (op-object)
    `(let ((,op-object
            (setf (gethash ,name *dsl-operators*)
                  (make-instance 'dsl-operator
                                 :name ,name
                                 :arity ,arity)
                  )))
       (when (or,prod-jumps ,dm-jumps)
         (register-jump-locations ,name
                                  ,prod-jumps
                                  ,dm-jumps))
       (setf (compiler-for ,op-object)
             ,compiler-for
             )
       (setf (productions ,op-object)
             ,productions
             ))))
```

Listing 17: Define Operator Macro

Moving away from declarative memory generation to production generation, the actual process of generating productions for an operator is handled by the *define-operator* macro in Listing 17. It is a light wrapper for the normal object constructor for *dsl-operator* which performs a modified version of De Brujin Indexing (De Bruijn, 1972) on the jump location tables to transform the named operator-specific jump locations into anonymous integer values in $\mathbb{N}_0$, which are unique jump identifiers within the scope of that operator. This is done to avoid name collisions among operators, since great care is needed to do that with the declarative memory elements in a global way, but a restricted scope is needed here. As a result, no global uniqueness management needs to be done at this step (which reduces overhead significantly). Aside from managing jump location identifiers, the macro also provides a location to write the code for productions by hand.

While hand-written code is fine for some uses (e.g. *recenter-hands*), it is not good practice to repeat code unnecessarily. As a result, the actual productions are written as Common Lisp "backquote" template forms, where parts of the template are computed at compile-time, and interpolated or spliced into the template body. This allows for many design idioms to be expressed without needing to be manually written out, as well as allowing for standardized handlers to be developed.

Two notable examples of these mechanisms are the I/O interaction generators, and the *argument-p-sequence-for*[54] function. Both of these functions take information about the desired operation, and spit out code specialized to handling that portion of the code's behavior. By adhering to a common calling convention, both the code which would lead to running the generated productions, as well as the code which would be the recipient of any return values from the generated productions can be treated generically—increasing resusability and composability.

When an I/O interaction generator is called, such as *motor-keystroke-literal-generator* which has the type:

$$(\mathbb{Z} \to \mathbb{Z} \to [production]) \to vert? \to zero? \to [production] \tag{3.11}$$

Where the first argument is function which takes a pair of integers, and returns a list of productions, the second argument is a boolean to indicate whether the vertical values are fixed or parametric, and the last argument is a

[54] Its code is available upon request.

boolean to indicate if the numbering when generated should start at 0 or 1. When called, it effectively uses the passed argument as a templating system, and will run the template for each possible position the mechanical action should take. These generated values also include pre-generated information about where on the screen or keyboard a particular element is located, thus bypassing the need to compute that during runtime by calling out to a *!bind!*; both this and *!eval!* are standard parts of ACT-R, and allow for the embedding of Common Lisp code into the ACT-R production rules, with some nuances covered in the ACT-R manual. Some uses of these idioms can generate hundreds of productions, parameterized by both the row, column, and problem size. With all three, the function uniquely determines the location of a particular item on the screen[55]. Writing these kinds of productions by hand would be prohibitively time consuming, and in order to avoid doing so may lead to a change in experimental design. Instead, there is no change needed to make the real GUI for humans the same as the one for the ACT-R models.

Unlike the parametric templating system used in in *motor-keystroke-literal-generator*, *argument-p-sequence-for* represents a different approach to producing code. It does not take a template and make many instances of it, instead it takes the details for an operator and creates an entire set of behavior handlers which would otherwise be repeated ad nauseam for each and every operator. Its type is:

$$opsym \rightarrow opstr \rightarrow argcount \rightarrow argnum \rightarrow littype? \rightarrow control? \rightarrow [production] \tag{3.12}$$

Here it takes an operator symbol and string label, along with how many args are there and which argument it is, and the literal type as well as a boolean to indicate a control operator, and returns a list of productions. The code returned by *argument-p-sequence-for* is especially interesting, as it includes both generic operator loading handlers used by most operators to perform jumps and loads, as well as type-specific call handlers. These later ones automatically perform a rehearsal of an appropriate value when handling either a literal or the return value of a subexpression evaluation. As well, it also handles storing the results of the correct argument in the right place in the memory elements representing the operator, generating additional *op-sequence* chunks as needed. By having a function to call when this behavior was needed, it became possible to coordinate common behaviors across all relevant operators, but while also maintaining uniqueness of code paths among the ACT-R productions.

The compiler entry point is when `ast-compile!` is called upon the set up but not compiled `*compiler-instance*` variable. This is where the actual compiler object resides in the system, and it is mutated by several preparatory steps to accurately contain the inputs and other control metadata in one place.

```
(defun ast-compile! (compiler-state)
  (let* ((raw-input (input-ast compiler-state))
         (unwrapped-input (cdr (if (listp raw-input)
                                   raw-input
                                   nil)))
         (_ignorethis (progn
                        (set-main-loop-cutoff-length! unwrapped-input)

                        ))
         (root-name "root")
         (root-operator (gethash root-name *dsl-operators*))
         (root-compiler (compiler-for root-operator))
         (compiler-dm-output
           (funcall
            root-compiler
            compiler-state   ;state
            unwrapped-input  ;args
            nil              ;parent-symbol
            'root            ;my-symbol
            nil              ;return-symbol
            nil              ;return-state
            nil              ;return-op
            nil              ;parent
            nil              ;parent-arg-number
            ))
         (actr-formatted-dm-output
           (mapcar
            #'(lambda (dmop)
                (let*
                  ((dmop-args (args dmop))
                   (arglen (length dmop-args)))
                  `(,(if (dm-identity-override dmop) (dm-identity-override dmop) (branch-name dmop))
```

[55]These three values are needed due to the face that the position of a particular row or column is partly determined by the length of the problem. The length of the problem matters because the columns are center-aligned rather than fixed left.

```
                ISA op-sequence
                branch-name ,(branch-name dmop)
                branch-order ,(branch-order dmop)
                done         ,(or (done? dmop) :empty)
                return-branch ,(return-branch dmop)
                return-state  ,(return-state  dmop)
                return-operator ,(return-operator dmop)
                op-name ,(if (stringp (name dmop)) (intern (string-upcase (name dmop)))  (name dmop))
                arg0    ,(if (< 0 arglen) (aux-normalize-strings (elt dmop-args 0)) :empty)
                arg1    ,(if (< 1 arglen) (aux-normalize-strings (elt dmop-args 1)) :empty)
                arg2    ,(if (< 2 arglen) (aux-normalize-strings (elt dmop-args 2)) :empty)
                arg3    ,(if (< 3 arglen) (aux-normalize-strings (elt dmop-args 3)) :empty)
                arg4    ,(if (< 4 arglen) (aux-normalize-strings (elt dmop-args 4)) :empty)
                arg5    ,(if (< 5 arglen) (aux-normalize-strings (elt dmop-args 5)) :empty)
                arg6    ,(if (< 6 arglen) (aux-normalize-strings (elt dmop-args 6)) :empty)
                timestamp     :no-value ; :empty
                problem       :no-value ; :empty
                last-argument :no-value
                loop-iteration :no-value
                       ))
                       )
                       compiler-dm-output))
      (compiler-productions-output
       (loop for k being the hash-keys of *dsl-operators*
          unless (equalp k root-name)
          appending (productions (gethash k *dsl-operators*))))
      (input-sgp-arguments *SGP-ARGS-ALIST*)
      (problems *problem-list*)
      (final-ast (compile-wrapper actr-formatted-dm-output compiler-productions-output input-sgp-arguments
      ↪  problems))
      )
(setf (output-ast compiler-state) final-ast)
compiler-state))
```

Listing 18: Ast-Compile! Listing

The final crucial portion of the compiler lives within the `compiler-sequence-for` function in Listing 30. While its detailed behavior can be gleaned from its source, it is the key to actually generating memory elements which represent the *Stored Program* part of the Von Neumann Architecture. It also makes certain that there are not name collisions and that each and every possible execution branch is represented in memory.

It divides up the processing based on operator arity cases. Zero arity is the simplest case, where the flags are set to indicate the return branch, state, and operator, as well as parent links. Higher arities work similarly, save that they also include DM elements to represent each of its arguments return values and processing steps.

3.3.6 Procedural Learning

One of the biggest hurdles facing the original embedded DSL design for this compiler was that its heavy use of `!bind!` and `!eval!` ACT-R operators precluded any productions which used them from being fair game for ACT-R's procedural learning mechanism, called *Production Compilation* or *chunking*. Essentially, it finds pairs of productions which fire sequentially, and generates a new production which does the work of both, but only has the overhead of a single production being fired.

According to the ACT-R reference manual (Bothell, n.d.), a production is disqualified for *Production Compilation* for any of the following reasons in addition to the usual compatibility considerations:

- is the time between the productions greater than the threshold time?
- does either production have a `!eval!` condition or action?
- does either production have a `!bind!` or `!safe-bind!` condition?
- does either production have a `!bind!` action?
- does either production use `!mv-bind!` in either the conditions or actions?
- does either production test the same buffer more than once in its conditions?
- does either production have a buffer overwrite action?
- does either production use a direct chunk request?

- does either production use slot modifiers other than `=` in its conditions?
- does the first production make multiple requests using the same buffer?
- does the first production have a RHS `!stop!` action?

As the embedded DSL design relied heavily on both `!bind!` and `!eval!`, there were multiple rules saying it could not model this kind of learning. In order to avoid that shortfall, the new compiler design relies upon `!bind!` and `!eval!` as little as possible, doing as much work as practical using pure production rules generated by the compiler.

Preliminary testing of the new design showed that *Production Compilation* occurred very readily, and the expected kinds of learning could be detected. Unfortunately, the fine details of the implementation of *Production Compilation* caused it to be more aggressive than expected when compiling productions. It would not only combine them in a straightforward way, but it would also aggressively assume that the Retrieval Buffer could be pruned from the combined production. Since the Retrieval Buffer is normally *internal* to the model, and cannot be changed externally, the *chunking* algorithm treats it specially. For most modeling needs, this is probably fine, but for the models produced by the compiler, it is occasionally too aggressive an optimization. When run, there are occasions when the compilation window width (ACT-R's control variable for this is `:tt`, which defaults to 2 seconds) and the productions usage of the retrieval buffer resulted in errors occurring where the buffer was cleared when it shouldn't have been, or it would drop a retrieval buffer modification request from the RHS that should have been left alone.

All of this effort begs the question of why should this compiler care about *Production Compilation*. The answer, besides the obvious desire to be compatible with both the ACT-R theory and implementation, is that this mechanism of learning can be used to account for quantitative differences between the models generated by this compiler and those written by hand. As was explained earlier, implementing a Turing-complete computer based on Stored-Program von Neumann Architecture introduces seemingly unaccountable amounts of overhead in the form of production calls and memory accesses tied to managing a Call Stack. Surely, any models made in this way would be slower and use more memory than their optimized hand-written equivalents? Perhaps, and is it not also the case that many of the published models and quantitative calibration of the ACT-R architecture are all based on these optimized models? Both are true. However, that does not mean that the Call Stack system that this compiler uses can be rejected outright on the grounds of quantitative differences.

The reasoning for this is nuanced: if *Production Compilation* reduces both the number of productions evaluated as well as the number of redundant memory accesses, and thereby increases the speed and reduces the memory overhead of the model, then it is possible that the same mechanism might apply to the overhead of the Call Stack as well. If such were the case, then these models of a much more powerful computer would be able to potentially converge towards the performance of the optimized hand-tuned models. This is not a philosophical question, but rather one that is purely practical and testable. If such was the case, the performance curve predicted by the ACT-R theory of learning would suggest that the learning curves of both kinds of model should be of the same kind, but with the optimized hand-tuned models quantitatively showing more "expertise"[56] or "training time" when initially run; for the same amount of run-time the general model should always lag behind in its "expertise".

Should this be proven true, then Call Stack models may be a method for generalizing much of the modeling work that has been done in the past. If it works as described, models would be more composable, could be reused for other models, and the kinds of problems which are considered fair game for being modeled would widely broaden. Compilation systems like the one used for this compiler could likewise allow a new level of analysis, focusing on the high-level behaviors of humans in the description of their behavior, but without loosing quantitative predictiveness. Thus, this issue is one which is very much of relevance.

Because both the undesirable behavior of the *Production Compilation Module* when interacting with the productions generated by this compiler, and the very important role that learning via this mechanism plays in this modeling effort, steps have been taken to fix this behavior. At present the solution is to make use of what I dub "safety annotations" which are the inclusion of harmless no-op checks on all operators, where most ACT-R productions are modified as so: This change causes the *Production Compilation Module* to interpret the production as being in one of the categories that it is not allowed to compile, without changing any other semantics (careful analysis of this was made for each edit). Doing so does not generally disable all learning, and it does not prevent other non-modified productions from being compiled. Instead it is a very carefully considered and conservative compromise between allowing errors versus disallowing learning entirely, or changing the semantics of the models.

Other options that present themselves for addressing these issues include a number of changes that are quite outside of the scope of this work. Modification of the *Production Compilation* candidate detection algorithm could allow for a clearer way of avoiding problematic production rules in the first place. Similarly, the compilation modules could also be modified to perform a less aggressive pruning optimization, which does not lead to dropped operations

[56] In this usage, "expertise" is a problem-specific performance metric that exhibits the classic ACT-R learning behavior in humans.

```
(p
   ;; ... actual LHS...
   ?retrieval>
   state free
   - state error
   ==>
   ;; ... actual RHS...
)
```

Listing 19: ACT-R Buffer Ready Test Template Example

without at least correcting the corresponding detection code. There are other options as well, which would require this work to already be implemented, and then used to test for post hoc applicability. For example, all operators could be completely rewritten in such a way as to avoid potentially problematic (but otherwise licit) buffer utilization. A similar effort which includes buffer drop correction may also work. Finally, parametric analysis of the ACT-R *Production Compilation* window width (i.e. `:tt`) may yield results.

One of the interesting conclusion of this work is based on these observations, namely that ACT-R's theory does not forbid learning which would treat interacting with safe memory elements, but its implementation lacks the means to do so without pruning out those very memory elements. The implication with this is explored in greater depth elsewhere in the Results and Future Work sections, but its worth mentioning at this point, to provide context. Should a mechanism be improved, it is possible the marked overhead that the generic computational model has could be compiled into something closer in performance to a real human or a hand-optimized model.

This work suggests that it should be possible, though the production compilation may need to be able to apply a safety check before compilation can proceed, verifying that elements are all held constant or loaded in the same order each time via a limited kind of branch execution tracing, as many JIT compilers do. Rather uniquely, this work already provides a static version of branch labeling through its branch analysis and automatic symbol generation used in generating productions for operators. Adjusting the Production Compilation Module to include this information for scanning purposes could be a starting point for any future implementation of this mechanism that interacts with a DSL like this one. Even if not used for DSL design, there might be a semi-manual method of annotating or decorating productions or memory elements with this kind of metadata.

3.4 Operator Modeling

With the preliminary definition of terms and basic operator design goals complete, this section focuses instead on the design decisions that lie behind the actual modeling of these operators using ACT-R. Such reporting is commonplace in other works utilizing cognitive architectures, so it is included here to provide the other half of the domain standard of reporting model and model-products (e.g. trace data) together. While the compiler used in this work is surely larger than most models, it and the actual ACT-R models it produces can both be similar analyzed structurally. If nothing else, it grounds discussions of the implementation without requiring a recourse to the roughly 27000 lines of raw source code.

This section contains the actual implementation and source code for all of the DSL's operators, as well as a handful of special operators used for certain niche purposes, but not otherwise generatable by the GE Phenotype expansion, and thus unreachable. These operators are vital to understanding and grounding the DSL's choice of behaviors to model in terms of actual ACT-R production rules. Thus, by the theoretical foundation behind ACT-R, each operator is an executable theory about how a person does that activity.

While there are blocks of source code here, they are excerpts from the larger body of the compiler, and are relatively self-contained. Where Operators are structurally similar outside of type, they will be presented together with only one of the types code listing being presented fully. Any special behavior based solely on the type difference will be presented in terms of the few changes to the code that may be relevant, so long as it is clear. Those operators which are massively different from each other based on type will not be presented in this way.

It should be noted that the first few Operators in this section are undoubtedly the most important. At least, they are the most important insofar as they are the first Operators which present universal features common to all (non-Special) Operators, and so they contain detailed explanations for these universal features, which are not repeated elsewhere. Specific Operators to look at include Literals in Section 3.4.1 for information about the type system, MAIN in Section 3.4.2 for information about the looping wrapper, ROOT in Section 3.4.3 for information about the generation and labeling of branches as well as the memory elements that represent them, RECENTER-HANDS in Section 3.4.4 for information about normal (non-Special) Operators and motor actions, READ-WHOLE in Section

3.4.5 for information about visual and memory actions, and the vital SHIFT-HAND in Section 3.4.6 which introduces the real details of how (non-zero arity) Operators are applied to their arguments and return values. The final important ones are SWAP in Section 3.4.9 for the details of how actual UI interaction work as well as the information about processing more than one argument without losing intermediate return values, and in IF-N is Section 3.4.10 in which Flow Control Operators are introduced.

3.4.1 Literals

While the numeric, character, and boolean literals are not technically Operators, they are included here to clarify their role in the implementation of Operators. Simply put, some values in Common Lisp evaluate to themselves, that is they are constant literal values. The DSL Compiler keeps this practice for consistency sake, and performs case-based analysis when performing evaluations of AST expressions, to correctly identify one of the small handful of pre-known literals.

They are treated specially by `argument-p-sequence-for`, for example. Instead of needing to recurse upon them like a normal Operator, they are detected and loaded into the correct slots, as though they had actually run a nameless Operator which stuck that literal value into the `return-value` slot of the METAPROC within the goal buffer. The calling Operator is ignorant of the internal implementation of those children/arguments it calls, and so it does not know that it can skip certain memory loading behaviors that it might otherwise not need to do if a child was not fully called.

It is perhaps relevant to say here that the string inputs of a value Block Sorting DSL program cannot legally contain embedded Common Lisp expressions by design–it supplies no `eval`. Thus, while these literals may ultimately be grounded in Common Lisp, they are not strictly speaking the same types. For example, the DSL Compiler's `:type-number` is fully enumerated in the Block Sorting BNF grammar, and is not actually Common Lisp type `NUMBER` which is the superclass of the numeric tower within Common Lisp[57].

The same treatment is given to `:type-letter`, which is not actually Common Lisp type `CHARACTER`, even though they look like they could. As well, although they are encoded as strings of length exactly one, they are not Common Lisp type `STRING`. They are a different type because they are values within a language built on top of Common Lisp and ACT-R, but which has its own semantics, despite using the same syntax (at least to the extent defined in the grammar).

Finally, boolean literals are of type `:type-boolean`, and are not the standard Common Lisp values of `T`, `NIL`, `'()`, nor any form of Common Lisp generalized boolean values. Instead, they are just the symbols `'true` and `'false`, and no other Common Lisp symbols are permitted in that type class. Other Block Sorting DSL values are not *truthy/falsey*[58] like they are in Common Lisp.

This consistent differentiation from the host language's types is purposeful. This language is Nominally Typed, not Structurally Typed, or Duck Typed. That is, just because the implementation details of a type share common data-structures or behavioral properties, does not mean that they are *the same type*. As well, there is no casting of any sort in this language, so numbers are numbers and characters are characters, and there is no recourse to `char-code` or `code-char` to convert freely between them. As well, this is a Statically typed language, and the compiler explicitly knows the types of all parts of the program ahead of time.

While there may be some Operators which are implemented in terms of looking up a (relatively) constant value, those values are not Literals, and are evaluated like any other Operator.

3.4.2 MAIN Special Operator

Before any Stored Program can be run at all, there needs to be a initial loading system. Modern Computers have a boot loader that jumps to the OS. Programs from the C family of programming languages have a named entry point that the OS targets for loading, called `main` which is usually wired up to the OS-dependent executable format (e.g. ELF binaries on Linux, or PE binaries on Windows).

By analogy with those, the DSL Compiler prepares its own special target for bootstrapping a running Block Sorting DSL program. This one is called `main` but its name is neither actually special, like in C, nor is it accessible via BNF expansion, neither is it a named Operator within the system. Instead, the DSL Compiler implements it by injecting the code in Listing 31 after the expanded declarative memory elements in `,@dm-list` but before the actual expanded production rules from the inputs Block Sorting DSL program string, in `,@p-list`.

[57] This includes: Complex, Real, Float, Rational, Integer, Ratio, Fixnum, Bignum, and several kinds of specific precision floating point types, per the Hyperspec.

[58] A term used in the C-family of languages as well as many others for types which can be rather trivially converted to a boolean value automatically by the compiler or runtime.

It begins with `main-start-timer`, which initializes the internal time perception for the model, as well as ACT-R level timer. The execution will then loop between checking for being done, loading the next part of the stored program, and seeing if we are ever ready to react to completion or iteration or break times. Actual completion of a single correct sorting problem solution will cause a tone to be played, and the model uses this for notification as well, lacking any special knowledge of the GUI's internal state. The looping behavior here is the implementation of the iterative portion of the `with-dsl-wrapper` behavior. It is also one of the very few places where actual direct Common Lisp code is called, mostly for math.

This code is initialized through the combination of the `starting-state` METAPROC instance in Listing 53, and the command `(goal-focus starting-state)` being run before the system is told to start the experiment.

3.4.3 ROOT Special Operator

In order for there to be a sole Abstract Syntax Tree representing a valid DSL program, there needs to be one and only one rooted Tree. As a matter of course, having the DSL able to generate several sequential steps from the base BNF grammar, as well as having the wrapper stick in the utility step of initially reading the current problem, would result in fact a Forest of unattached Trees, each with their own roots. Those that design data structures have long corrected this by reducing Forests into a single rooted Tree by introducing an artificial extra node that all otherwise unrelated Trees are made children of. In our case, this is done so as to preserve the same evaluation order that the DSL normally uses–keeping sequential steps in lock-step.

This ROOT special operator does not even have an actual name or representation in the grammar, rather it is only an artifact of the DSL Compiler, under the name `root`. Since it is special, there is a limit of exactly one `root`. It is also unique in that it is the only operator of arity `:varargs`. All other operators have a fixed predetermined artiy. It may be viewed as being the implementation of the `with-dsl-wrapper` macro.

```
(define-operator
    :name "root"
    :arity :varargs
    :compiler-for
    #'(lambda (compiler-state
               args
               parent-sym my-sym
               return-sym return-state return-op
               parent parent-arg-number
               )
        (let* ((root-ident
                (car
                 (register-idents
                  '(root)
                  compiler-state)))
               (arg0-literal? (not (listp (car args))))
               (root-dm-object nil)
               (argcount (length args))
               (argcount-1 (1- argcount))
                  (root-dm-objects (make-array (1+ argcount) :initial-element nil))
               (res nil)
               )
          (setf *first-branch-id* root-ident)
         (setf root-dm-object
                (make-instance
                 'dsl-op-sequence
                 :name  "root"
                 :branch-name  root-ident
                 :branch-order 0
                 :arity :varargs
                 :args  nil ;args
                 :done? (< argcount 2)
                 :return-branch   :empty
                 :return-state    :empty
                 :return-operator :empty
                 :parent          :empty
                 :parent-argument-number :empty
                 :dm-identity-override 'root0
                 )
                )
          (setf (elt root-dm-objects 0) root-dm-object) ;root is special
             (loop for i from 1 upto argcount
```

```
   do
     (progn
       (setf (elt root-dm-objects i) ;make extra branch-orders
             (make-instance
              'dsl-op-sequence
              :name  "root"
              :branch-name  root-ident
              :branch-order i
              :arity :varargs
              :args  nil ;args
              :done? (= i argcount);(= i (1- argcount))
              :return-branch   :empty
              :return-state    :empty
              :return-operator :empty
              :parent          :empty
              :parent-argument-number :empty
              :dm-identity-override (intern (format nil "ROOT~d" i))
              ))) )
(loop
   for subexpr in args
   with current-arg-num = 0
   do
     (progn
      (if (not (listp subexpr))
           ;;literal detected, skip the rest of this
           (setf (args-literal-value (elt root-dm-objects current-arg-num)) subexpr)
           ;;else, non-literal, perhaps with 0-args?
           (let* (
                  (op-sym  (car subexpr))
                  (op-sym-normalized-string (dsl-symbol-to-string op-sym))
                  (op-args (cdr subexpr))
                  (op-args-len (length op-args))
                  (zero-arity-operator? (zerop op-args-len))
                  (op-def-obj (gethash  op-sym-normalized-string *dsl-operators* ))
                  (op-compiler-fn (compiler-for op-def-obj))
                  (arg-ident
                   (car
                    (register-idents
                     (list op-sym)
                     compiler-state)))
                  (dm-items
                   (if zero-arity-operator?
                       (funcall ; 0-arity still needs evaluation
                        op-compiler-fn
                        compiler-state
                        op-args
                        root-ident arg-ident
                        root-ident
                        (1+ current-arg-num)
                        'root
                        root-ident current-arg-num)
                       (funcall
                        op-compiler-fn
                        compiler-state
                        op-args
                        root-ident arg-ident
                        root-ident
                        (1+ current-arg-num)
                        'root
                        root-ident current-arg-num)))
                  )

             ;; then store them
             (setf res (append res dm-items))
             (pop-ident compiler-state)
             (let* ((entry-point-for-subexpr (first dm-items))
                    (op (dsl-string-to-symbol (name entry-point-for-subexpr)))
                    (branchop (branch-name entry-point-for-subexpr))
                    (branchorder (branch-order entry-point-for-subexpr))
                    (current (elt root-dm-objects current-arg-num))
                    )
               (setf (return-operator current) op)
```

```
                    (setf (return-branch   current) branchop)
                    (setf (return-state    current) branchorder)
                    )

                  ))
            (incf current-arg-num)
            )
          )
      ;;append there at the front to keep readability
      (setf res (append (map 'list #'identity root-dm-objects) res))
      ;;return a list of dsl-op-sequences as follows: let
      ;;MYLIST be the parts just for this element, in the order
      ;;that they appear here, and ARGLISTS
      ;;be the lists returned by the CHILDARGS then return the
      ;;following list: (append MYLIST ,@ARGLISTS )
      res)))
```

Listing 20: ROOT Special Operator Listing

Through this code block, the all important execution graph is traced out and each branch is labeled. Once these are set up, they are stitched together within the other operators, so that the flow of control is aware of the parent-child relationship between an operator and its arguments.

3.4.4 RECENTER-HANDS

Perhaps the simplest operator in the DSL, `recenter-hands` contains nothing superfluous or external, and is a great introduction to how the flow of control works.

Each operator has a entry point, which its parents will load for it by setting the current branch, branch order, and operator to match it. Once that is done, the retrieval buffer is used to request an op-sequence which represents it's part of the Stored Program from long term memory.

Once that is done, most operators–including this one–will start to do some of their actual work. Those with arguments will generally evaluate them at this point. As a zero arity operator, there is nothing to be done of that sort, and so the operator uses the manual buffer to request the relevant hand movement.

It uses the branch order slot in the goal buffer's METAPROC instance to sequence its individual steps. Each LHS carefully makes sure to uniquely follow the RHS of its immediate predecessor production.

When it is finally done, the return value is usually passed back to its caller/parent via the return-value slot of the goal buffer. Since this operator never returns any values, it does not touch the return value. Control is passed back to its caller/parent by setting the current branch, branch order, and operator to that which `compiler-sequence-for` added into the DM elements when compiling the Stored Program out of the input Block Sorting DSL program string.

```
(define-operator
    :name "recenter-hands"
    :arity 0
    :compiler-for (compiler-sequence-for 'recenter-hands "recenter-hands" 0)
    :productions
           (list
        '(p recenter-hands
          =goal>
          ISA metaproc
          current-branch =branch
          branch-order   0
          operator       recenter-hands
          ?retrieval>
          state free
          ==>
          +retrieval>
          ISA op-sequence
          branch-name  =branch
          branch-order 0
          op-name      recenter-hands
          =goal>
          branch-order 1
          )
        '(p recenter-left
          =goal>
          ISA metaproc
          current-branch =branch
```

```
  branch-order   1
  operator       recenter-hands
  return-value   =return-value
  =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
  return-state  =return-state
  return-operator =return-op
  op-name       recenter-hands
  ?manual>
  state free
  ?retrieval>
  state free
  - state error
  ==>
  +manual>
  ISA point-hand-at-key
  hand left
  to-key 4
  =retrieval>
  =goal>
  next-branch        =return-branch
  next-branch-number =return-state
  return-value       :no-value
  branch-order       2
  )
'(p recenter-right
  =goal>
  ISA metaproc
  current-branch     =branch
  branch-order       2
  operator           recenter-hands
  next-branch        =return-branch
  next-branch-number =return-state
  =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
  return-state  =return-state
  return-operator =return-op
  op-name       recenter-hands
  ?manual>
  state free
  - state error
  ==>
  =retrieval>
  +manual>
  ISA point-hand-at-key
  hand right
  to-key 7
  =goal>
  branch-order 3
  )
`(p recenter-done
  =goal>
  ISA metaproc
  current-branch     =branch
  branch-order       3
  operator           recenter-hands
  next-branch        =return-branch
  next-branch-number =return-state
  =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
```

```
        return-state  =return-state
        return-operator =return-op
        op-name       recenter-hands
        ?manual>
        state free
        - state error
        ==>
        ,@(log-return-value "recenter-hands" :no-value)
        =goal>
        current-branch     =return-branch
        branch-order       =return-state
        operator           =return-op
        dm-reload  :reload
        subgoal    :empty
        )
      )
  )
```

Listing 21: 0-Arity Recenter Hands Listing

3.4.5 READ-WHOLE

This is the only operator which is not strictly speaking necessary. Its raison d'être has been touched upon a few times already, but to paraphrase them: it was the most clearly observed experimental behavior, and would be unreasonably complex to evolve through random heuristic chance, so it was included.

It is arguably the most complex zero arity operator, combining looping, visual behaviors, and memory behaviors. The most important parts of this break down into a simple set of rules. It can be found in Listing 32.

First, for whatever size the current problem is, the `screen-pos-literal-generator` maps the correct combination of length, row, and column to a unique production which matches it, and then sets the visual system to acquire visual data from that place.

Second, after looking at that place, the visual information is encoded into declarative memory elements of type `letters` in the production `read-whole-lookat-encode`.

Third, after years of experimentation and testing, a rehearsal is added to cause statistical preference to the most recent transient location information. This is a single rehearsal, but it is possible to repeat the Operator to rehearse it again. This rehearsal is located in the production, `read-whole-lookat-rehersal`.

Fourth, should the current slot not be the last one, the production will return to the `screen-pos-literal-generator` productions, due to `read-whole-lookat-rehersal-cleanup` setting `dm-reload` to `:empty` without modifying anything else. Due to the initial `screen-pos-literal-generator` setting their next iteration when they initially execute, with only that slot value not being `:empty` to trigger their firing back to their next slot.

Finally, once the iterations have read all there is to see, the system returns `:no-value` in the end of the `read-whole-return` production rule. It uses the same techniques as already shown in `recenter-hands` to determine its caller/parent, and return its value to them. This value is essentially $\perp$ (the symbol for the value Bottom in most type systems), and is not very useful on its own, being little more than an official out-of-band value to flag error states. This production, and most other ones that have side effects—which are motor, visual, or memory actions—are not pure functions, and are not guaranteed to be idempotent.

3.4.6 SHIFT-HAND

Building on the motor and memory actions of `recenter-hands`, `shift-hand` is the first Operator that actually includes the processes for handling a non-zero arity. It has exactly one argument of `:type-number`, which semantically represents which of the number keys on the keyboard (not the number-pad keys, the top row of number keys on a standard QWERTY layout) to strike or press; in this case dwell above, ready to strike the key beneath the hanging finger.

Referring to `argument-p-sequence-for` function in Listing 54, the code generator is designed to return a list of production rules which are customized to fit within the gap between the productions `shift-hand` and `shift-hand-arg0-done` within Listing 33, where they are stuck in place of the generator call.

In these productions the number of the arguments are 0-indexed, just like in Common Lisp or C, so the zeroth argument is the first subexpression (e.g. `(+ 0 1 2)`, 0 is the zeroth argument index, which is the first argument) of the S-Exp encoding for the Operator. This is important for keeping track of sequential behaviors within operators with multiple arguments, and is used for all Operators with more than zero arguments.

Passing control to the type-agnostic `all-prepare` production from `argument-p-sequence-for` is the start of passing control to the uniform argument/subexpression handling code in the DSL Compiler. Here in the RHS of the `shift-hand` production, it can be seen using the retrieval buffer to request an `op-sequence` element specific to this combination of operator, branch, and branch-order value (as defined in the `:dm-jumps` as `'dm-recall-arg0`), as well as setting the goal buffer's METAPROC's subgoal to `:empty` and its branch-order to the one unique to recalling arg0 (as defined in the `:prod-jumps` as `'recall-arg0`).

This will be matched by the generated `all-prepare` method for this code path, uniquely. From there it will either jump to `all-recall-subexpr` or `all-recall-literal`, depending on whether the input was determined by the DSL Compiler to contain a literal expression or a subexpression. If it was a literal value, then that argument's value in the loaded `op-sequence` would be a value besides `:empty`, specifically that literal constant. Otherwise, the empty value there indicates that there is a subexpression that needs to be loaded and jumped to.

There are two paths the code might take here. For a literal, `all-recall-literal` is followed by `all-recall-literal-save-literal` then `all-recall-literal-commit`. All uses of the imaginal buffer are followed by a *-commit* production rule, and these are no different. Thus, the creation of a new `op-sequence` element specific to this combination of operator, branch, and branch-order value, and combinations of arg-N slots, as well as a comprehensive timestamp/last-argument/loop-iteration/problem uniqueness marker. The arg-N slots are filled with `:empty` when they have not been processed, and either `:no-value` or the actual stored return values of subexpressions that have already been computed.

That might sound like overkill, but the combination of all of those permits the most `op-sequence` elements to be uniquely identified with a particular point in the program's execution—which is important during dynamic looping—causes potentially unpredictable activation of some stored`op-sequence` elements. In this case, they are being computed in the RHS of `all-recall-literal-save-literal`. This will be expanded upon in SWAP, in Section 3.4.9, which needs to store the results of evaluating arg0 while it evaluates arg1, or else it would never be able to perform its behaviors.

Once `all-recall-literal-commit` has cleared the imaginal buffer, it passes control back to the original Operator again, by requesting the retrieval buffer to recall an `op-sequence` with the unique combination of your branch-name, branch-order (per `,(prod-jump-fn 'dm-recall)`), and the current operator; as well as setting the goal buffer's subgoal to `:empty`, dm-reload to `:reload`, and branch order (per `,(prod-jump-fn 'return)`). These values are uniquely matched by `shift-hand-arg0-done`, where the Operator continues its own execution.

Now, the other code path is that of a non-literal argument, back in `all-recall-subexpr`. Its main behavior is to copy the `op-sequence` the return-branch, return-state , and return-operator slots loaded for that particular argument of that operator for that point the in the execution of the program. From the LHS they are copied into the RHS into the goal buffer, into current-branch, branch-order, and operator, respectively. The old value in the retrieval buffer is purged automatically by ACT-R since there is no request to retain it (i.e. no `=retrieval>` or `+retrieval>` in the RHS).

By doing this, the Operator passes control to another Operator, determined by the branch-order of 0, and a matching operator name. Once called, the Operator's execution only knows which parts of the Stored Program to load and run by the passed in current-branch, which is attached to the declarative memory elements called `op-sequence` by the DSL Compiler when it generates them at compile time. Dynamically created `op-sequence` are used too, but contain only the passing transient information about the return values of subexpressions, so that they can be computed in the correct order, and then used by the Operator. Similarly, every Operator passes control back to their caller/parent by looking up who their parent was within the `op-sequence` elements that the DSL Compiler generates for them all.

These operations can be seen in `shift-hand` and `shift-hand-really-return` productions, respectively. While simple cases allow the direct processing of a return-value in parent, it is the case that higher arity Operators naturally overwrite the METAPROC return-value slot in the goal buffer, since all Operators return their result (at least `:no-value`) using that same slot. This is analogous behavior for CPUs to store their results in a common register; in order to retain that value, it needs to be stored either in working memory (i.e. METAPROC slots) or in long term memory, via memory elements standing in for a storage tape or disk.

The actual work that is done with this evaluated subexpression's return-value is pretty minimal at this point, with special case handling for which hand will need to be moved, based on the value held in the return-value slot of the METAPROC in the goal buffer. The keys on the number-row go from one to nine, and then zero. So unless the slot contains zero, the keyboard is divided in half, with $[1, 5]$ being pressed by the left hand, and $[6, 9]$ being pressed by the right hand. The special case of the key "0" is handled as representing the value 10, and so it is handled by the right hand too.

Once a target key is identified, and one of the hands is selected to press that key, the keypress is requested. Provided it completes, the Operator has no special value to report, but the semantics of the language require that it

return `:no-value` if that is the case, so it is placed into the return-value slot of the METAPROC in the goal buffer, before the Operator returns to its caller/parents.

3.4.7 LOOK-OFF-SCREEN

This is the only one of several possible *fidgeting* Operators, chosen because it was one of the few that was easily verified visually during experiments with live subjects, as well as one of the few which has a reasonable theoretically and quantitatively grounded implementation available within ACT-R.

The operations within the Operator shown in Listing 34 are not that new. It takes one number argument which is the range of columns to randomly choose among, and then it randomly chooses a row and column to look at. After a short random delay based on the internal temporal buffer, the randomly selected coordinates on screen (only legal locations, not arbitrary screen coordinates) are looked at.

The temporal buffer and the `!bind!` clauses in the RHS of several production rules, are the only new things of note. The temporal buffer is a nuanced internal perception of time for an ACT-R model, and it functions like a clock, not a timer. So, implementing a timer in terms of the clock requires some light math which ACT-R is not easily able to do. For this purpose, some of that math is done within Common Lisp, and the resulting values are bound to an ACT-R variable with `!bind!` clauses. The use of extra-theoretical calls to the Common Lisp host for ACT-R (i.e. `!eval!` and `!bind!`) is minimized in the DSL Compiler, because they preclude procedural learning via production compilation.

3.4.8 ONCE-ONLY

This is one of the rarely used Operators–as seen in Listing 35–which is only used for its side-effects, which is to memoize the evaluation of a `:type-letter` expression. This memoization persists for the duration of the ACT-R model, after which the elements in declarative memory are purged, resetting the mechanism by which this memoization works. The Operator itself is more or less a Thunk which wraps its argument, before evaluating it once when called, and then only ever returning the value `:no-value` (not the original return value, since this is meant for side-effects only).

There are two code branches within this Operator, which are predicated upon either a successful retrieval or a unsuccessful one, in the `once-only` production rule's RHS. Successful recall lands in `once-only-recall-past-succcess`, while failure winds up in `once-only-recall-past-failure`.

The failure pathway is fairly normal for a 1-arity Operator, save for the added use of the imaginal buffer in `once-only-return-encode`. In that rule, the result gets saved off as a declarative memory element of type `op-sequence` with a `:no-value` in arg0, and whatever the original return-value of the subexpression was is discarded while it returns `:no-value`.

Where the failure pathway has already been run once, the success pathway will only ever run with all subsequent invocations. This causes an alternate return pathway to run, which starts with `once-only-recall-past-succcess`, and then `once-only-recall-past-succcess-lookup-parent` returns to the caller/parent without evaluating anything further. It again returns `:no-value`.

3.4.9 SWAP

While the SWAP Operator in Listing 36 may be vital for the function of any sorting activity that would happen, and it is in fact uniquely required within the Block Sorting BNF Grammar, it is not actually a Special Operator. Rather, it is a normal 2-arity Operator, which is very important for the functions the system is designed to model. Centrality makes this Operator a convenient place to handle motor actions along with their rehearsal, as well a suitable location to place moderate error handling for experimental user inputs. The first important thing to examine here, is how the 1-arity handling of storing intermediate return values is built upon to handle 2-arity (or higher, though we don't have too many of those) Operators.

Looking at `swap-arg1-done` it is possible to see a few key traits that let SWAP know that it is recalling the `op-sequence` memory element that correctly has the results of its two arguments evaluated. First, `swap-arg1-done` is not entered on the LHS with any `op-sequence` besides the same one it would always be called with. Instead, it the RHS retrieval request for not only the multipart timestamp, but also both arguments being `- :empty`, as well as the `last-argument 1` which means that the last completed argument was the second of two arguments, indicating completion.

Once this request has been made, the request either winds up matching `swap-prep-coords`, or one of the rules that matches inputs of "0". Rules like `swap-prep-coords-bad-*` match error cases instead. Any illegal request is

turned into either a noop, or is bounded by the nearest upper/lower boundary value that key can be pressed for that length of problem.

The motor actions are actually very close to how SHIFT-HAND was implemented, and won't be repeated. Similarly, the rehearsals from READ-WHOLE is similar to the behavior that the SWAP rehearsals use, save that each of the new positions is rehearsed three times each. Experimental results showed that this provided a reasonable recall likelihood, compared to older positional information.

After the rehearsals are done, the Operator returns `:no-value`.

3.4.10 IF-N and IF-C

This is the first section in which two Operators are being discussed jointly, and may be seen in Listing 37. IF-N and IF-C differ only in their typing, with the first handling `:type-number` branches, while IF-C handles `:type-letter` instead. The sole reason for them to be separated is because the grammar requires monomorphic functions only, elsewise a more conventional polymorphic macro like Common Lisp's `if` might work.

Building on SHIFT-HAND and SWAP, dealing with 3-arity Operators is straightforward, just with each step being handled in sequence. Or rather, it would be handled completely sequentially, save that IF-N is a Flow Control Operator, so that the calls to `argument-p-sequence-for` are as follows:

0. `,@(argument-p-sequence-for 'if-n "if-n" 3 0 :type-boolean)`
1. `,@(argument-p-sequence-for 'if-n "if-n" 3 1 :type-number nil t)`
2. `,@(argument-p-sequence-for 'if-n "if-n" 3 2 :type-number nil t)`

This can be interpreted as follows, the first argument handler is a normal `:type-boolean` which is always evaluated. However, both of the other paths pass in the two extra optional arguments to `argument-p-sequence-for`, `nil` and `t`. The first is still the default value of nil for the `control-op` optional parameter of that function, which is actual an override value to place instead of normal evaluation (with the default of `nil` causing it to be ignored). The second, `t`, enables the normally disabled `path-control-operator`, which replaces the normal subexpression results saving behavior with one that is able to fill in `:no-value` for unevaluated arguments, as though they had been evaluated and produced no useful value. This is required in order to permit uniform argument handling among all non-zero arity Operators, including Flow Control Operators.

Given that, the IF-N Operator behaves identically to Common Lisp's `if` Special Operator[59], save that it always has a mandatory *else-form* rather than optionally having one. Based on the return value of the first argument's boolean test, it will either call the subexpression of the second argument (if true) or the third argument (if false). Whichever completes, all of the results are noted (including fake noop results for the unevaluated path) and the final return value of the IF-N is take from the result of the real evaluated path, and passed back to IF-N's parent/caller via the return-value slot of the METAPROC in the goal buffer.

3.4.11 THEN-N and THEN-C

The simplest of the BNF `<control-num-expr>` expressions, THEN-N (and THEN-C, which again differs only in type) is not part of an IF-THEN pair, but rather it is a fixed arity two version of the Common Lisp `progn` operator[60]. The normal `progn` is of variable arity, but evaluates all of its enclosed forms, and then returns the value of the last one; THEN-N does this for a fixed pair of two `:type-number` expressions. The name was chosen to sound like "...and then do this..." at a glance.

While it is only called for the purpose of side-effect for its first argument, it can be used to bind sequential behaviors, and can in fact be self-nested, creating something not dissimilar to `progn` in its application. The implementation of this is unsurprising for a two-arity Operator, and it merely discards the first return value while returning the second.

```
(define-operator
    :name "then-n"
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
```

[59] n.b. This is the Common Lisp Hyperspec's Special Operator, not the DSL Compiler's Special Operator, the DSL Compiler's use of the term is in reference to its host language's usage. Common Lisp's usage is more broadly called Special Forms, which includes Special Operators as a subset.

[60] Again, this is a Common Lisp Special Operator, like `if` discussed above.

```
                return-arg0
                save-arg0
                done-arg0
                recall-arg1
                jump-arg1
                return-arg1
                save-arg1
                done-arg1
                do-operation
                return-from)
:dm-jumps    '(dm-recall-arg0
                dm-recall-arg1
                dm-recall-parent
                )
:compiler-for (compiler-sequence-for 'then-n "then-n" 2)
:productions
    `(
      (p then-n
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator      then-n
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "then-n" 'dm-recall 0)
         op-name      then-n
         =goal>
         branch-order ,(jump-state-lookup "then-n" 'recall 0)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'then-n "then-n" 2 0 :type-number)

      (p then-n-arg0-done
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "then-n" 'done 0)
         operator       then-n
         return-value   =returnval
         =retrieval>
         ISA op-sequence
         branch-name    =branch
         branch-order   ,(jump-state-lookup "then-n" 'dm-recall-parent)
         op-name        then-n
         return-branch  =return-branch
         return-state   =reutrn-state
         return-operator =return-op
         ?retrieval>
         state free
         - state error
         ?imaginal>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "then-n" 'dm-recall 1)
         op-name      then-n
         =goal>
         branch-order ,(jump-state-lookup "then-n" 'recall 1)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'then-n "then-n" 2 1 :type-number)

      (p then-n-arg1-done
```

```
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(jump-state-lookup "then-n" 'done 1)
    operator       then-n
    return-value   =return-value
    starting-order =starting-order
    loop-iteration =loop-iteration
    timestamp      =timestamp
    =retrieval>
    ISA op-sequence
    branch-name    =branch
    branch-order   ,(jump-state-lookup "then-n" 'dm-recall-parent)
    op-name        then-n
    return-branch  =return-branch
    return-state   =reutrn-state
    return-operator =return-op
    ?retrieval>
    state free
    - state error
    ?imaginal>
    state free
    ==>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(jump-state-lookup "then-n" 'dm-recall-parent)
    op-name      then-n
    done           t
    - arg0         :empty
    - arg1         :empty
    arg2         :empty
    timestamp      =timestamp
    loop-iteration =loop-iteration
    last-argument  1
    problem        =starting-order
    =goal>
    branch-order ,(jump-state-lookup "then-n" 'return-from)
    return-value =return-value
    )

(p then-n-return
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(jump-state-lookup "then-n" 'return-from)
    operator        then-n
    return-value   =returnval
    =retrieval>
    ISA op-sequence
    branch-name    =branch
    branch-order   ,(jump-state-lookup "then-n" 'dm-recall-parent)
    op-name        then-n
    done           t
    - arg0         :empty
    - arg1         :empty
    arg1           =arg1
    arg2           :empty
    return-branch  =return-branch
    return-state   =return-state
    return-operator =return-op
    ?retrieval>
    state free
    ==>
    =goal>
    current-branch =return-branch
    branch-order   =return-state
    operator       =return-op
    return-value   =arg1
    arg0           :empty
    arg1           :empty
    dm-reload      :reload
```

```
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )

       )

     )
```

Listing 22: THEN-N Operator Listing

3.4.12 ONCE-PER-PROBLEM-N and ONCE-PER-PROBLEM-C

The final BNF <control-num-expr> expression, seen in Listing 38, ONCE-PER-PROBLEM-N (and ONCE-PER-PROBLEM-C, which again differs only in type) is nearly identical to the behavior of ONCE-ONLY, save that the reset policy is tied to each new problem, rather than the duration of the ACT-R Model.

Resetting in this way is handled simply within the production rule once-per-problem-n which requests not just any saved result, but the saved result for the current problem's original ordering, as stored in the starting-order slot of the METAPROC in the goal buffer. This slot is automatically updated when loading a new problem, and so makes old saved values unreachable,

3.4.13 MOST-RECENT-INDEX and MOST-RECENT-LETTER

This pair of operators don't just differ by type, but they also have most of their internals in common as well. Both use the normal ACT-R activation bias to recall any number or letter (respectively). Other than that free recall criteria, there is no other methods used to narrow the selection, so it is the least constrained recall of all of the similar recall operators.

Its purpose is to provide a way to just get a familiar number or letter for algorithmic purposes, perhaps for activation-biased semi-random selection.

```
(define-operator
:name "most-recent-index"
:arity 0
:compiler-for (compiler-sequence-for 'most-recent-index "most-recent-index" 0)
:productions
(list
 '(p most-recent-index
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   0
   operator       most-recent-index
   subgoal        :empty
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order 0
   op-name      most-recent-index
   =goal>
   branch-order 1
   subgoal :recall
   )
 '(p most-recent-index-prepare
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   1
   operator       most-recent-index
   return-value   =return-value
   starting-order =startorder
   length         =len
   timestamp      =timestamp
```

```
  subgoal        :recall
  =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
  return-state  =return-state
  return-operator =return-op
  op-name       most-recent-index
  ?retrieval>
  state free
  - state error
  ==>
  +retrieval>
  ISA letters
  =goal>
  branch-order       2
  )
'(p most-recent-index-match-good
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   2
  operator       most-recent-index
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :recall
  =retrieval>
  ISA letters
  letter        =letter
  slot-number   =slot
  ?retrieval>
  state free
  - state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       most-recent-index
  =goal>
  branch-order       3
  subgoal            :return
  return-value       =slot
  )
'(p most-recent-index-match-bad
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   2
  operator       most-recent-index
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :recall
  =retrieval>
  ISA problem-instance
  starting-order =startorder
  length         =len
  ?retrieval>
  state free
  state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       most-recent-index
```

```
   =goal>
   branch-order       3
   subgoal            :return
   return-value       :no-value
   )

 `(p most-recent-index-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       3
   operator           most-recent-index
   subgoal            :return
   return-value       =returnval
   =retrieval>
   ISA op-sequence
   branch-name   =branch
   branch-order  0
   done          =done
   return-branch =return-branch
   return-state  =return-state
   return-operator =return-op
   op-name       most-recent-index
   ?retrieval>
   state free
   ==>
   =goal>
   current-branch     =return-branch
   branch-order       =return-state
   operator           =return-op
   dm-reload  :reload
   subgoal    :empty
   )
 )
)
```

Listing 23: MOST-RECENT-INDEX Operator Listing

3.4.14 NOTED-INDEX and NOTED-LETTER

This pair of Operators differ only in return types. They each require that NOTE-INDEX or NOTE-LETTER have been called, or else they will fail to recall. Unlike the other memory lookup Operators, these look at the special chunk type(s): `number-instance` (or `letter-instance`).

These types are more or less the normal `letters` type, save that they have a label of `:noted` and the problem and ticks fields that connect to the problem and current time perception (which is not the same as the real timestamp, but is a consciously updated timer that is copied periodically).

Unlike most other Operators, these try to specify more closely which noted index or letter by both activation level, and a high-water search for the highest number of ticks for that problem. This means that its not a fool-proof recall method, but an approximate one. This intentionally tries to balance the utility of being able to reliably recall something that the experimental subject committed to memory, but which could have interference from other prior runs and recalls. This is separate from other visual/motor rehearsal behaviors, and does no explicit rehearsal.

```
(define-operator
    :name "noted-index"
    :arity 0
    :compiler-for (compiler-sequence-for 'noted-index "noted-index" 0)
    :productions
    (list
        '(p noted-index
          =goal>
          ISA metaproc
          current-branch =branch
          branch-order   0
          operator       noted-index
          subgoal        :empty
          ?retrieval>
          state free
```

```
  ==>
  +retrieval>
  ISA op-sequence
  branch-name  =branch
  branch-order 0
  op-name      noted-index
  =goal>
  branch-order :first-lookup
  subgoal :recall
  )
'(p noted-index-prepare
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   :first-lookup
  operator       noted-index
  return-value   =return-value
  starting-order =startorder
  length         =len
  timestamp      =timestamp
  subgoal        :recall
   =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
  return-state  =return-state
  return-operator =return-op
  op-name       noted-index
  ?retrieval>
  state free
  - state error
  ==>
  +retrieval>
  ISA number-instance
  label :noted
  problem =startorder
  =goal>
  branch-order       :iter-match
  arg0               =timestamp
  )

'(p noted-index-iter-match-bad
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   :iter-match
  operator       noted-index
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :recall
  ?retrieval>
  state free
  state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       noted-index
  =goal>
  branch-order       :clean-up
  subgoal            :return
  return-value       :no-value
  )

'(p noted-index-iter-match-start
  =goal>
  ISA metaproc
```

```
   current-branch =branch
   branch-order   :iter-match
   operator       noted-index
   return-value   =return-value
   starting-order =startorder
   length         =len
   timestamp      =timestamp
   loop-iteration =loop-iteration
   subgoal        :recall
    - arg0  :empty
   arg0    =highwater
   =retrieval>
   ISA number-instance
   value =value
   label :noted
   problem =startorder
   ticks =ticks
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA number-instance
   label :noted
   problem =startorder
   > ticks =ticks
   =goal>
   branch-order       :iter-match
   arg0               =ticks
   subgoal            :iterate
   return-value       =value
   )
'(p noted-index-iter-match-check
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   :iter-match
   operator       noted-index
   return-value   =return-value
   starting-order =startorder
   length         =len
   subgoal        :iterate
    - arg0  :empty
   arg0    =highwater
   =retrieval>
   ISA number-instance
   value =value
   label :noted
   problem =startorder
   ticks =ticks
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA number-instance
   label :noted
   problem =startorder
   > ticks =ticks
   =goal>
   branch-order       :iter-match
   arg0               =ticks
   subgoal            :iterate
   return-value       =value
   )

'(p noted-index-iter-match-exhausted
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   :iter-match
   operator       noted-index
```

```
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :iterate
   - arg0  :empty
  arg0    =highwater
  ?retrieval>
  state free
  state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       noted-index
  =goal>
  branch-order       :clean-up
  subgoal            :return
  )

`(p noted-index-cleanup
  =goal>
  ISA metaproc
  current-branch     =branch
  branch-order       :clean-up
  operator           noted-index
  subgoal            :return
  return-value       =returnval
  =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
  return-state  =return-state
  return-operator =return-op
  op-name       noted-index
  ?retrieval>
  state free
  - state error
  ==>
  =goal>
  current-branch     =return-branch
  branch-order       =return-state
  operator           =return-op
  arg0       :empty
  dm-reload  :reload
  subgoal    :empty
  ;subgoal :reload
  )
)
   )
```

Listing 24: NOTED-INDEX Operator Listing

3.4.15 INDEX-OF-LETTER and LETTER-OF-INDEX

This pair of operators are paired not because they are the same save for return value, but instead because they are inverse operators. Applying them to each other cancels their results out (similarly to $(2X)/2$).

The basic use of these operators is to recall from memory the corresponding slot that the model recalls for a particular letter (for INDEX-OF-LETTER), or the reverse of that (for LETTER-OF-INDEX). This is not to say that the results are always representative of the truth in the UI, but this operator gains reliability only in the presence of rehearsals (like in SWAP).

Unlike so many of the `letters` chunk type recall operators, which are specially designed to ignore declarative memory elements which represent obsolete states which not only do not need recalled, but which would be actively misleading if they were considered for normal activation-based recalls. These use the composite timestamp that has been mentioned elsewhere.

Instead, these are the only way to perform a `letters` lookup without any kind of additional restrictions, other than the single parameter being matched (i.e. slot or letter).

```
(define-operator
    :name "index-of-letter"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from)
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    )
    :compiler-for (compiler-sequence-for 'index-of-letter "index-of-letter" 1)
    :productions
    `(
      (p index-of-letter
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator       index-of-letter
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "index-of-letter" 'dm-recall 0)
         op-name      index-of-letter
         =goal>
         branch-order ,(jump-state-lookup "index-of-letter" 'recall 0)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'index-of-letter "index-of-letter" 1 0 :type-letter)

      (p index-of-letter-arg0-done
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "index-of-letter" 'done 0)
         operator       index-of-letter
         subgoal        :empty
         return-value   =return-value
         ?retrieval>
         state free
         - state error
         =retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order  ,(jump-state-lookup "index-of-letter" 'dm-recall-parent)
         op-name      index-of-letter
         return-branch    =return-branch
         return-state     =return-state
         return-operator  =return-op
         ?imaginal>
         state free
         ==>
         =retrieval>
         =goal>
         current-branch     =branch
         branch-order       ,(jump-state-lookup "index-of-letter" 'done 0)
         operator           index-of-letter
         subgoal   :lookup
         dm-reload :empty
         )

      (p index-of-letter-lookup
```

```
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "index-of-letter" 'done 0)
   operator       index-of-letter
   return-value   =return-value
   subgoal        :lookup
   dm-reload      :empty
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA letters
   letter =return-value
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "index-of-letter" 'done 0)
   operator           index-of-letter
   subgoal   :match
   dm-reload :empty
   )

(p index-of-letter-match-good
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "index-of-letter" 'done 0)
   operator       index-of-letter
   return-value   =return-value
   subgoal        :match
   dm-reload      :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA letters
   slot-number =slot-num
   letter      =return-value
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "index-of-letter" 'dm-recall-parent)
   op-name      index-of-letter
   done         t
   arg0         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "index-of-letter" 'done 0)
   operator           index-of-letter
   return-value       =slot-num
   subgoal   :load-done
   dm-reload :empty
   )

(p index-of-letter-match-bad
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "index-of-letter" 'done 0)
   operator       index-of-letter
   return-value   =return-value
   subgoal        :match
   dm-reload      :empty
   ?retrieval>
   state free
   state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "index-of-letter" 'dm-recall-parent)
```

```
   op-name      index-of-letter
   done         t
   arg0         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "index-of-letter" 'done 0)
   operator           index-of-letter
   return-value       :no-value
   subgoal   :load-done
   dm-reload :empty
   )

(p index-of-letter-load-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "index-of-letter" 'done 0)
   operator       index-of-letter
   subgoal        :load-done
   dm-reload      :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "index-of-letter" 'dm-recall-parent)
   op-name      index-of-letter
   done         t
   arg0         :empty
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   -imaginal>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "index-of-letter" 'return-from)
   operator           index-of-letter
   subgoal   :empty
   dm-reload :reload
   )

(p index-of-letter-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "index-of-letter" 'return-from)
   operator       index-of-letter
   return-value   =return-value
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "index-of-letter" 'dm-recall-parent)
   op-name      index-of-letter
   done         t
   arg0         :empty
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   =goal>
   current-branch     =return-branch
   branch-order       =return-state
```

```
        operator          =return-op
        dm-reload :reload
        subgoal   :empty
        )

    )
)
```

Listing 25: INDEX-OF-LETTER Operator Listing

3.4.16 NEXT-NUMBER and PREV-NUMBER

This is another pair of operators that are inverses of each other, seen in Listing 39. They otherwise have the same argument and return types.

Unlike some of the others letter or number operators which deal with actual `letters` declarative memory elements, these perform a precompiled direct lookup. There is the capacity for error handling to be performed, detecting out-of-bounds requests and returning $\perp$ for `:type-number`, `:no-value`.

Within the production rules, all of the pre-compiled matchers match legal keyboard number row keys, and produce their return-value by manually coded mapping. In this system, the "0" key represents the value 10, so $ord(9) < ord(0)$, letting $ord(x)$ be the ordinal value of x. The only difference between these two operators is which direction the manual mapping points the successor value.

3.4.17 NEXT-LETTER and PREV-LETTER

This is another pair of operators that are inverses of each other. They otherwise have the same argument and return types, and can be seen in Listing 40.

Unlike some of the others letter or number operators that deal with actual `letters` declarative memory elements, these perform a structural traversal of the chunk type `alpha-order`, which encode the alphabet for our purposes. This encoding and design of `alpha-order` can be seen in Listing 7. What is relevant here is that this chunk type encodes the alphabet as a doubly-linked list, permitting bidirectional traversal. This predicts linear-time, $O(n)$, lookups. There is also an ordinal value encoded therein, but it is not used to cheat with random-access in this application.

Beginning with `next-letter-find-start` which looks up the first `alpha-order` chunk by ordinal index (this and the opposite lookup from PREV-LETTER are the only times it is used). This is the head of the doubly-linked list, and it is traversed one cell at a time (n.b. these cell are not Common Lisp *cons* cells, but rather an DSL Compiler-level type, reified as an ACT-R chunk type of our own).

Traversal proceeds linkwise, walking the linkage that represents the list until either the newly loaded cell contains the letter to match as its letter slot, or we run out of cells and return `:no-value` to signal an error. In order to reverse this operation, the tail of the list is looked up, and then walked in the reverse order. They are otherwise identical.

3.4.18 SCAN-FOR-CHAR-LR and SCAN-FOR-CHAR-RL

While all of the SCAN-FOR-* Operators share their internals with each other, they are all designed such that they actually do not look up values from memory, but instead cause visual buffer actions to occur. That is they perform an actual scan of the UI for either the number or letter being requested. The LR or RL suffix indicates the direction of traversal, either left-to-right or right-to-left, respectively.

These two are paired in Listing 41 because their type signatures are identical, and other than the direction of traversal, they are quite similar. Functionally, they are meant to look up the current location of a input letter, without needing to refer to memory activation biased chunks, which can be misleading.

3.4.19 SCAN-FOR-NUM-LR and SCAN-FOR-NUM-RL

Everything that has been said for SCAN-FOR-CHAR-LR (and -RL) apply here, aside from the reversed type signature– and can be seen in Listing 42. The function of this Operator is to return the letter that is found in a particular slot value. As well, it does not permit random-access lookup of a location, only linear lookup again[61].

[61] See LOOK-AT-CHAR in Listing 43, if you want random access, at the expense of potentially being out of date with the current UI due to memory actions or motor actions

3.4.20 NOTE-INDEX and NOTE-LETTER

These operators are paired because they differ only in two parts: return type, and their internal chunk type for representing their NOTE-* behavior. To begin with, these must be run before either of the NOTED-* Operators can be used without returning `:no-value`.

Once these Operators have been passed their argument, they store its evaluated return from the arg0 slot in the stored `op-sequence` produced by their argument0 evaluation. It is then copied into the return-value slot of the METAPROC in the goal buffer.

After storing it, the imaginal buffer is used to store either a `number-instance` (for NOTE-INDEX) or `letter-instance` (for NOTE-LETTER). This happens in `note-index-note-start` where the value slot stores the return-value. The label slot is also assigned `:noted` value for reference, and the composite timestamp is used by other Operators which care about recency and obsolescence.

```
(define-operator
    :name "note-index"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from
                     do-note
                     really-return
                     )
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    )
    :compiler-for (compiler-sequence-for 'note-index "note-index" 1)
    :productions
    `(
      (p note-index
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator       note-index
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "note-index" 'dm-recall 0)
         op-name      note-index
         =goal>
         branch-order ,(jump-state-lookup "note-index" 'recall 0)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'note-index "note-index" 1 0 :type-number)

      (p note-index-arg0-done
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "note-index" 'done 0)
         operator       note-index
         starting-order =starting-order
         loop-iteration =loop-iteration
         timestamp      =timestamp
         ?retrieval>
         state free
         - state error
         =retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
         op-name      note-index
```

```
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
   op-name      note-index
   - arg0       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  0
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "note-index" 'return-from)
   operator           note-index
   dm-reload :reload
   )

(p note-index-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "note-index" 'return-from)
   operator       note-index
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
   op-name      note-index
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "note-index" 'do-note)
   operator           note-index
   return-value       =arg0
   subgoal            :empty
   )

(p note-index-find-start-fail
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "note-index" 'do-note)
   operator       note-index
   return-value   :no-value
   subgoal        :empty
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
   op-name      note-index
```

```
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
   op-name      note-index
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "note-index" 'really-return)
   operator           note-index
   return-value       =arg0
   )

(p note-index-note-start
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "note-index" 'do-note)
   operator       note-index
   - return-value :no-value
   return-value   =matchthis ; get ordinal from lookup -> pick toplevel -> walk pointers linearly -> load pointer
   ↪  target TODO
   subgoal        :empty
   starting-order =originalorder
   timestamp      =timestamp
   loop-iteration =iteration
   time-since-process-start =ticks
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
   op-name      note-index
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   =temporal>
   ISA time
   ticks =realticks
   ?temporal>
   state free
   ==>
   =temporal>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
   op-name      note-index
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   +imaginal>
   ISA number-instance
   value =arg0
   label :noted
   problem =originalorder
```

```
      iteration =iteration
      ticks =realticks
        =goal>
      current-branch     =branch
      branch-order       ,(jump-state-lookup "note-index" 'really-return)
      operator           note-index
      )

    (p note-index-really-return
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order   ,(jump-state-lookup "note-index" 'really-return)
      operator       note-index
      - return-value :empty
      return-value   =return-value
      starting-order =originalorder
      time-since-process-start =ticks
      ?imaginal>
      state free
      =retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order  ,(jump-state-lookup "note-index" 'dm-recall-parent)
      op-name      note-index
      done         t
      arg0         :empty
      arg1         :empty
      arg2         :empty
      return-branch    =return-branch
      return-state     =return-state
      return-operator  =return-op
      ?retrieval>
      state free
      - state error
      ==>
      =goal>
      current-branch     =return-branch
      branch-order       =return-state
      operator           =return-op
      dm-reload :reload
      subgoal   :empty
      )

    )
  )
```

Listing 26: NOTE-INDEX Operator Listing

3.4.21 LOOK-AT-CHAR and LOOK-AT-NUM

These Operators are paired in Listing 43 because they differ only in their types being reversed. Both of them use a memory recall operation to try to recall a `letters` instance, and then perform visual buffer requests to look at the previously saved position.

As has already been hinted at in Listing 42, these Operators are the only actual random access Operators which interact with the UI. They are included because it is not at all unreasonable for a real person to look at a particular place or letter from memory, but in practice and experimentally, their utility is marginal.

Because there is no special knowledge of the UI's state without visual buffer requests, the only way to have random access is to store locations in memory chunks and perform retrieval buffer requests for them. Activation can be effected strongly by other Operators, and so whether or not the value being requested is recent enough to still accurately reflect the UI's state is strongly dependent on the sequence of operations which have already happened when these Operators are called.

As a result of the UI's design, the slots indexes are invariant, and so LOOK-AT-NUM is consistent, other than the returned letter value may be more recent than any entry current in memory. LOOK-AT-CHAR however suffers from the fact that the SWAP Operator mutates the UI state whenever it runs, perhaps causing inaccurate information by virtue of the fact that the old memory elements are not removed from memory.

3.4.22 FIRST-INDEX and FIRST-LETTER

This pair of Operators share everything other than their actual return type. Both of them require that the READ-WHOLE Operator has run at least once for the current problem.

So long as there are memory elements that encode the first slot, they will work normally. While both load up any valid `letters` chunks in memory, the major difference between them is that FIRST-INDEX reads that chunk and then always returns `1`. While, the other returns whatever letter the recalled chunk encoded.

Thus, FIRST-LETTER can be inaccurate, just like other memory lookups of `letters` chunks that don't worry about timestamps or similar things.

```
(define-operator
:name "first-index"
:arity 0
:compiler-for (compiler-sequence-for 'first-index "first-index" 0)
:productions
(list
 '(p first-index
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   0
   operator       first-index
   subgoal        :empty
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order 0
   op-name      first-index
   =goal>
   branch-order 1
   subgoal :recall
   )
 '(p first-index-prepare
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   1
   operator       first-index
   return-value   =return-value
   starting-order =startorder
   length         =len
   timestamp      =timestamp
   subgoal        :recall
   =retrieval>
   ISA op-sequence
   branch-name   =branch
   branch-order  0
   done          =done
   return-branch =return-branch
   return-state  =return-state
   return-operator =return-op
   op-name       first-index
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number   1
   =goal>
   branch-order       2
   )
 '(p first-index-match-good
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   2
   operator       first-index
```

```
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :recall
  =retrieval>
  ISA letters
  letter        =letter
  slot-number   1
  ?retrieval>
  state free
  - state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       first-index
  =goal>
  branch-order       3
  subgoal            :return
  return-value       1
  )
'(p first-index-match-bad
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   2
  operator       first-index
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :recall
  ?retrieval>
  state free
  state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       first-index
  =goal>
  branch-order       3
  subgoal            :return
  return-value       :no-value
  )

`(p first-index-cleanup
  =goal>
  ISA metaproc
  current-branch     =branch
  branch-order       3
  operator           first-index
  subgoal            :return
  return-value       =returnval
  =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
  return-state  =return-state
  return-operator =return-op
  op-name       first-index
  ?retrieval>
  state free
  ==>
  =goal>
  current-branch     =return-branch
  branch-order       =return-state
```

```
   operator           =return-op
   dm-reload  :reload
   subgoal    :empty
   )
 )
)
```

Listing 27: FIRST-INDEX Operator Listing

3.4.23 LAST-INDEX and LAST-LETTER

These Operators are fundamentally identical to FIRST-INDEX and FIRST-LETTER, save for being reversed. As a consequence of this, the return values are not consistent for LAST-INDEX, which is potentially going to recall the different actual numbers, which are based on the current problem length. This is tied into the length slot in the METAPROC within the goal buffer in `last-index-prepare`.

```
(define-operator
:name "last-index"
:arity 0
:compiler-for (compiler-sequence-for 'last-index "last-index" 0)
:productions
(list
 '(p last-index
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   0
   operator       last-index
   subgoal        :empty
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order 0
   op-name      last-index
   =goal>
   branch-order 1
   subgoal :recall
   )
 '(p last-index-prepare
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   1
   operator       last-index
   return-value   =return-value
   starting-order =startorder
   length         =len
   timestamp      =timestamp
   subgoal        :recall
   =retrieval>
   ISA op-sequence
   branch-name   =branch
   branch-order  0
   done          =done
   return-branch =return-branch
   return-state  =return-state
   return-operator =return-op
   op-name       last-index
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number   =len
   =goal>
```

```
  branch-order       2
  )
'(p last-index-match-good
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   2
  operator       last-index
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :recall
  =retrieval>
  ISA letters
  letter        =letter
  slot-number   =len
  ?retrieval>
  state free
  - state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       last-index
  =goal>
  branch-order       3
  subgoal            :return
  return-value       =len
  )
'(p last-index-match-bad
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   2
  operator       last-index
  return-value   =return-value
  starting-order =startorder
  length         =len
  subgoal        :recall
  ?retrieval>
  state free
  state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          t
  op-name       last-index
  =goal>
  branch-order       3
  subgoal            :return
  return-value       :no-value
  )

`(p last-index-cleanup
  =goal>
  ISA metaproc
  current-branch     =branch
  branch-order       3
  operator           last-index
  subgoal            :return
  return-value       =returnval
  =retrieval>
  ISA op-sequence
  branch-name   =branch
  branch-order  0
  done          =done
  return-branch =return-branch
  return-state  =return-state
```

```
   return-operator =return-op
   op-name       last-index
   ?retrieval>
   state free
   ==>
   =goal>
   current-branch     =return-branch
   branch-order       =return-state
   operator           =return-op
   dm-reload  :reload
   subgoal    :empty
   )
 )
)
```

Listing 28: LAST-INDEX Operator Listing

3.4.24 CURRENT-PROBLEM-LENGTH

This is quite possibly the simplest actual Operator, taking no arguments, performing no memory or io actions besides loading its own return `op-sequence`. Instead, it merely copies the length slot value from the METAPROC instance within the goal buffer to the return-value slot of the same. This slot is automatically updated by the DSL Compiler's mechanisms outside of this Operator, so it doesn't need to worry about very much.

```
(define-operator
:name "current-problem-length"
:arity 0
:compiler-for (compiler-sequence-for 'current-problem-length "current-problem-length" 0)
:productions
(list
 '(p current-problem-length
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   0
   operator       current-problem-length
   subgoal        :empty
   length         =len
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name   =branch
   branch-order  0
   =goal>
   branch-order 1
   return-value =len
   subgoal :return
   )

 `(p current-problem-length-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       1
   operator           current-problem-length
   subgoal            :return
   return-value       =returnval
   =retrieval>
   ISA op-sequence
   branch-name   =branch
   branch-order  0
   done          =done
   return-branch =return-branch
   return-state  =return-state
   return-operator =return-op
   op-name       current-problem-length
   ?retrieval>
```

```
   state free
   ==>
   =goal>
   current-branch     =return-branch
   branch-order       =return-state
   operator           =return-op
   dm-reload  :reload
   subgoal    :empty
   )
 )
)
```

Listing 29: CURRENT-PROBLEM-LENGTH Operator Listing

3.4.25 NEXT-LETTER-IN-SONG and PREV-LETTER-IN-SONG

These Operators are almost identical to NEXT-LETTER and PREV-LETTER, aside from how they are implemented. In the best case, they are supposed to have a better performance on lookup times than the non-song-based Operators (which are linear-time lookups); see Listing 44.

This is accomplished by using the `alpha-song-chunk` type for its traversal, after performing a single ordinal lookup using an `alpha-order` to check which part of the tree structure should be retrieved first, and then performing a linear traversal within that portion of the song. The entry point into this is the production rule `next-letter-in-song-find-lookup-ord`.

What these Operators wind up doing is cleanly representing the fact that singing portions of the Alphabet Song (Klahr et al., 1983). The intuitive explanation is that a person who learned the alphabet with the song, as many children in the US do, are expected to recall portions of the melody and lyrics in segments. While the segments are random access, they only provide a way to bypass earlier segments of the song, and they are internally linear, using a similar list structure to NEXT-LETTER.

This results in first looking up the `alpha-song-chunk` being used to find which portion of the song needs to be recalled, and then within that portion, the letters are walked in the same way as the song does. Effectively, this allows for portions of the song to be skipped, and only the relevant portion traversed linearly.

3.4.26 AND

This is the first of the `:type-boolean` Operators, in Listing 45. As a group, they follow the same kind of general flow as most other Operators, despite the different type pathways. The only major difference is that there is no special chunk type in memory for this category of Operators to work on, nor are there any special NOTE-* type Operators either.

Within this Operator, the logic of the Boolean conjunction, $\wedge$, is implemented by division by cases. The relevant rules effectively implement a Truth Table for Boolean conjunction, save for the handling for `:no-value`. This is implemented beginning with `and-prep-args` rule. It loads only legal values, so as to simplify the Truth Table down to a conventional Boolean conjunction.

3.4.27 OR

This Operator as well AND and XOR are more or less identical aside from the Truth Table they implement, seen in Listing 46. See AND above for the introduction. This one implements Boolean disjunction, $\vee$.

3.4.28 NOT

This Operator is the only unary `:type-boolean` Operator; see Listing 47. It also implements a Truth Table, but it is a simple one, given that it has only one argument. It represents the Boolean negation, $\neg$.

3.4.29 XOR

This Operator is technically not needed, as XOR can be implemented through nested applications of the other Boolean operations; it is in Listing 48. It is included to reduce the complexity required to represent it, much like READ-WHOLE.

Again, this Operator is nearly identical to AND and OR, other than the details of the Truth Table implementation. This one is the Boolean exclusive disjunction, $\oplus$.

3.4.30 NUM< and CHAR<

These Operators are the second class of `:type-boolean` Operators, ordinal comparators; see Listing 49. Unlike the other `:type-boolean` Operators, these actual perform their function without a Truth Table. Instead, they make use of ACT-R's built in ordinal comparison feature in LHS matchers within production rules.

In the production rule `num<-prep-args`, the arguments are loaded for matching by case. The rules that follow take its output and match each of the possible cases, returning the correct output depending on the actual case that rule represented. Since ACT-R treats letters as ordinal types, the comparison logic for CHAR< is almost identical to NUM<

3.4.31 NUM> and CHAR>

These Operators are almost identical NUM< and CHAR<, save that their original matching logic is reversed; see Listing 50.

3.4.32 NUM= and CHAR=

These Operators are also nearly identical to the other ordinal comparison Operators, in Listing 51. The control flow is kept uniform with the other Operators, even though they could be reduced in complexity down to a simple ACT-R match. Part of this rationale is to keep the category as mutually consistent as possible, so that their impact on the generation of models would not vary wildly.

3.4.33 ASSERT-N and ASSERT-C and ASSERT-B Special Operators

These Special Operators–in Listing 52–are only used for diagnostic purposes, to implement Unit Testing for the system. Per the `when` form below, they are not even defined for the DSL Compiler unless both `*debug-mode*` and `*run-testing-harness*` are true. The only difference between these and the normal equality Operators is that this one halts immediately if the comparison fails. In that sense it works nearly identically to assertions in normal programming languages. Note, that Common Lisp's `assert` is distinct from Java or C's assertions, in that it is a *correctable error*, and can be *continued*. This version is not, and it halts after printing diagnostic information. When called in this way, all normal input ASTs are replaced with the testing input instead.

3.5 Grammatical Evolution

The control parameters and fitness functions used by the GP processes are described in this section. Parameters may not make much sense in isolation, but they can be provided upon request. There are no single set of them that make all problems equally solvable in the same time frame. It is sufficient to understand that the nature of the sampling process assumes that an experimenter will vary everything from random seeds to mutation rates and crossover methods. The only things that are fixed are the architectures themselves, and the BNF grammars, as well as the DSL Compiler.

3.5.1 Human-Matching Fitness Function

This is the fitness function for matching against human data.

x a chromosome trace

x_H a reference human trace

p is a penalty value, of 1000

$S(x)$ a function indicating success by returning 0 otherwise p

β a bias value in $[0, 1]$

C_t total number of problems

$C_c(x)$ number of correct problems; $0 \leq C_c(x) \leq C_t$

$\Delta_s(x, x_H)$ the string difference, per Bard (2007)

$\Delta_e(x, x_H)$ the non-uniform Euclidean Distance (see the code listing below)

k_s string difference scalar in $\mathbb{R}_0$

k_e euclidean distance scalar in $\mathbb{R}_0$

c_e a maximum euclidean distance penalty $\mathbb{R}_0$

n_H the keystoke length of the human trace in $\mathbb{R}_0$

n_G the keystoke length of the generated trace in $\mathbb{R}_0$

k_k keystroke length difference scalar in $\mathbb{R}_0$

c_k keystroke length difference penalty $\mathbb{R}_0$

r adjusted proportion of time in $[0, 1]$

$$f_H(x) = S(x) + \beta k_s \Delta_s(x, x_H) + (1 - \beta) k_e tanh(2\Delta_e(x, x_H)/c_e) + k_k tanh(2|rn_H - n_g|/c_k) + p(rC_t - C_c(x)) \quad (3.13)$$

As all fitness functions should be approximately linear in their scaling, as well as possess the ability to emphasize or deemphasize certain criteria, this fitness function is a simple weighted sum of factors. All fields have been re-scaled to be of commensurate scale[62] to the penalty value, p, as will be explained term by term. The first term $S(x)$ is on the order of p by definition, and so needs no special weighting. The second term, experimentally determined to be on the order of 100 gains a scaling factor which is normally set as $k_s = p$. The third term, $\Delta_e(x, x_H)$ returns unbounded $\mathbb{R}_0$ values, and so it is converted to the same scale of p via $k_e tanh(2\Delta_e(x, x_H)/c_e)$ where a calibration factor is used to re-scale a sigmoid function; $tanh$ is the base sigmoid used here for three reasons: it has only one degree of freedom (as opposed to five for the natural logarithm sigmoid), it naturally has an output range of $[0, 1]$ for all inputs, and it is part of the standard math libraries for most systems. The fourth and final term $C_t - C_c(x)$ is also experimentally on the order of 100, and so it also receives a p scalar. Finally, to support the shift of emphasis among terms, a bias term is introduced for the two non-error based components.

```
(defun non-uniform-euclidean-distance (a-raw b-raw &optional
         (window-width 1) (inputs-already-sorted nil))
  ;;Euclidean distance for non-uniform-dimensional vector.
  ;;The inputs must be lists of sorted numbers.
  ;;If not, destructive sorting will be done on the inputs.
  (let* ((a (if inputs-already-sorted a-raw (sort a-raw #'<)))
         (b (if inputs-already-sorted b-raw (sort b-raw #'<)))
         (diffs nil)) ;;stick the pairs' diffs here
    (do* ((as a) ;;we make these into stacks
          (bs b))
         ((not (and as bs)) ;;continue iterating until either runs out
          (if as ;;then stick the rest on...
              (dolist (extra-a as) (push extra-a diffs))
              (dolist (extra-b bs) (push extra-b diffs))
              )
          ) ;;and keep going until they're all empty,
            ;;returning nil because side-effects are all we need
      (let* ((ahead (or (car as) 0))
             (bhead (or (car bs) 0))
             (diff (- ahead bhead))
             (absdiff (abs diff))
             )
        (if (<= absdiff window-width)
            ;;good we can do something normal and real
            (progn
              (push diff diffs) ;store the diff
              (pop as) ;;remove the used values
              (pop bs)
              )
              ;;bad, we need to promote one into a 0 and
              ;;the leave the larger one for later.
              (progn
```

[62] That is within plus or minus one order of magnitude to p.

```
            (if (>= ahead bhead)
                (progn ;;work with b, save a for later
                  (push bhead diffs) ;;this doesn't need another subtraction
                  (pop bs)           ;;because (- b 0) = b
                  )
                (progn ;;work with a,
                       ;;save b for later, see above for math
                  (push ahead diffs)
                  (pop as)
                  )
                )
            )
          )
        )
      )
    ;; take the sqrt(sum(square(diff(pairs)))), diffs is already diffs(pairs)
    (sqrt ;;and we're done!
     (reduce #'+ ;;sum them
             (mapcar #'(lambda (x) (expt x 2)) diffs))) ;;square the diffs
    )
  )
```

3.5.2 Time-Optimizing Fitness Function

This is the fitness function for specifically optimizing total time to solve, and it does not use any information about human behavior, other than only choosing Operators from the DSL.

x a chromosome trace

p is a penalty value, of 1000

$S(x)$ a function indicating success by returning 0 otherwise p

C_t total number of problems

$C_c(x)$ number of correct problems; $0 \leq C_c(x) \leq C_t$

$T_t(x)$ the total time to complete the run, in milliseconds

r adjusted proportion of time in $[0, 1]$

$$f_T(x) = S(x) + T_t(x) + p(rC_t - C_c(x)) \tag{3.14}$$

To contrast the fitness function which performs trace matching, this one is much simpler and quite different in design. The only new term, $T_t(x)$ is experimentally on the order of 100000 for most runs, and so it is already on a similar scale of p. This design prefers correctness of problems to time, but does not rely on human data in any way.

Chapter 4

Results and Contributions

This chapter contains both the current state of the experimental designs that this research agenda has produced as well as a guideline for where immediate follow-on work would fit. Please refer to Figure 3.1 for the division between different applications or stages of the work. As well, Section 1.1 may be helpful as a contextual reference.

4.1 Human Experimentation

The modeling of individual experimental subjects as opposed to population average behavioral models, introduces certain design constraints upon the experiments that can be conducted.

While it is not strictly necessary, from a practical point of view, modeling individuals shifts the focus from the population average behavior to the extended behavior of each of several subjects. Resources being finite, this leads to the selection of a much smaller number of subjects, while each subject is used to produce many more data points per subject compared to most other kinds of similar experimental designs.

For statistical testing, the value of n is then no longer the number of individual humans in the experimental pool, but instead the number of problems or tasks they solve. Each trial is one datapoint on a time axis over the course of that individual's progress through the period of the experimental tasking. Thus, a single subject can provide hundreds or thousands of datapoints, depending on the nature of what is being measured. Having a secondary dataset from another individual (or a tertiary, etc.), is useful insofar as the experimenter is typically concerned with being able to model individuals, rather than just one specific individual.

Statistical power testing then compares against the null hypothesis of non-humans, which don't attempt to model humans in any quantitative way, against the actual humans and models that try to fit those individuals. This way, a poorly predictive quantitative model of an individual would be indistinguishable in its closeness of fit to those individuals' behaviors from a model, which does not model that person at all. Alternatively, models of other individuals could be used as the null hypothesis instead. The power testing in all these cases comes from the idea that models of specific individuals should more closely resemble those specific individuals, as compared to anything not directly modeling those individuals (e.g. a model of Alice should more closely match Alice than models of the population average, Bob, or Carol).

For individuals, especially in novel settings or with new tasks, there is a learning process. Indeed, data from early in an individual's experimental run may be erratic or they may not have learned or decided on an approach. That means that individual experiments with time course data require longer periods of exposure to hopefully capture the behaviors of the individual performing reliably in some quantitative way. This incentivizes further the use of longer experimental courses for a single individual.

It is almost automatic to assume that any kind of statistical analysis will require taking statistical averages from a population of interest. In fact, in modeling individuals, it is quite counter to that experience. Averaging over the population may in fact weaken the modeling process, insofar as an individual can be assumed to have some relatively self-consistent approach to problem-solving, as variable as it may be compared to other individuals. While there are indeed kinds of analyses that can be done once several individuals have been modeled, those analyses do not look for statistically average behavior, but instead looks at common tree structures instead of statistical elements. This inversion is due to the fact that models are generative and can be used to perform quantitative comparisons without reference to average behaviors.

Expanding on the premise that individual people generally have their own individual self-consistent approaches to problem solving, it then becomes the case the experimenter and the modelers have their own implicit methods. Whether or not these implicit methods are incorporated into the model, is a kind of bias, which can make it hard

for a modeler to consider more than a few options for their models, even though other humans may come up with a broader range. It is the problem of teleologic generation of models versus teleonomic generation of models. It at the very least implies that making the models directly by hand may not accurately capture or represent the range of possible behaviors to be modeled.

In order to permit modeling of individuals, it is advisable to take some care to select tasks that are broad enough to be interesting, but not so widely varied in the possible solutions as to make the modeling and representation process intractable. In this kind of work, algorithmic behaviors are the best when there are a range of options, and a sequence of datapoints to be generated by the individual. While it may be simpler to jump directly to solving the problems, it does make it harder to infer what happened in their heads while they were solving those problems.

Finally, it is the case that modeling average behaviors affords the protection of larger numbers against the variance of the individual. Here it is the intrinsically the opposite. Subjects get bored in long repetitive tasking. They can and do perform nonproductive thrashing behaviors. They can and do need to take breaks. Modeling needs to include this in their quantitative models in a sensible way. This work for example uses ACT-R's memory decay mechanisms to reflect the passage of time during a bathroom break or models looking off screen (say at their own fingers, to center or locate their positions), despite explicit instructions to the contrary.

4.1.1 Experimental Design

Applying the information discussed in the last section, this work is structured to take advantage of the longer experimental period and smaller number of test subjects. This begins with the selection of the exemplar task for grounding this research in a real task instead of only speaking abstractly.

As has been explained at length through this work, the exemplar task, the Block Sorting Task has been grounded in both human seriation work dating back over fifty years as well as in a similar history of research into sorting algorithms. What is highlighted in this section of the work, is how those issues explained above are addressed in its design.

Taking the basic design of early seriation experiments, the Block Sorting Task, is designed to make several recordable actions per problem solution so as to provide a behavioral Trace that includes not only the terminal actions, but also the intermediate states and timings exposed. For the purpose of evolving programs, it is best to have partial data to work with. It keeps the fitness landscape from being too flat and allows for a fitness gradient to be represented instead of locally flat stepwise landscapes.

As well, the limited number of options that the carefully minimized GUI presented also meant that the number of motor and visual actions that a real human would be able to make were likewise minimized. This permits the modeling efforts to focus on how the thoughts of the human user translated into those few keypresses or saccades, given the fact that whatever they were doing in their head took recordable amounts of time. This translates into models of the process, which can be expressed in the Block Sorting Grammar DSL program statements.

Those program statements are primarily about stringing together combined visual and memory actions, while ultimately pressing two keys. Those DSL programs can be arbitrarily complex within those boundaries, but the evolution will tend to higher fitnesses, which more closely resemble the sequence and timings of the actual individual being modeled.

In order to quantify the learning effects on the timings of the human Traces, this experimental design is made to include two categories of problems when it generates a test dataset. First, it makes random problems, of length four to ten, which are always out of alphabetical order (that is, it cannot legally generate an already solved problem instance), all of which contain the beginning of the alphabet in sequence, up till their problem length (so there are no missing letters, or repeated letters). The other category are identical, save for the fact that, once M of them are generated, they are repeated every C problems, so that the same problems are exposed repeatedly, in the same order.

This makes two learning curves to observe in the human's Trace data. For the repeater category, it can be said they occur multiple times in the same order each time, and they should have the upper-bound in learning. No other problems are likely to have as much of a time reduction as those M repeaters. The other category can only be judged based on a combination of that problems length, as well as its Levenshtein edit distance from the alphabet, as a kind of neutral complexity metric. This is needed to account for the fact that not all problems are of similar complexity, even if they are of similar length. For example "ABDC" is only one swap away from solved, while "DCBA" requires at least two.

Using both of these together, the limitations imposed on matching static (i.e. non-learning phase, in the long tail of the learning process, causing the experiment to discard the beginning data as unmodelable) human Trace data can be reduced or bypassed. As well, when combined with the minimization of anything which disrupts ACT-R production compilation, this makes it possible to model the learning in sequence with the whole human Traces. Similar mental operations will see similar learning curves for both categories, between the human and the model.

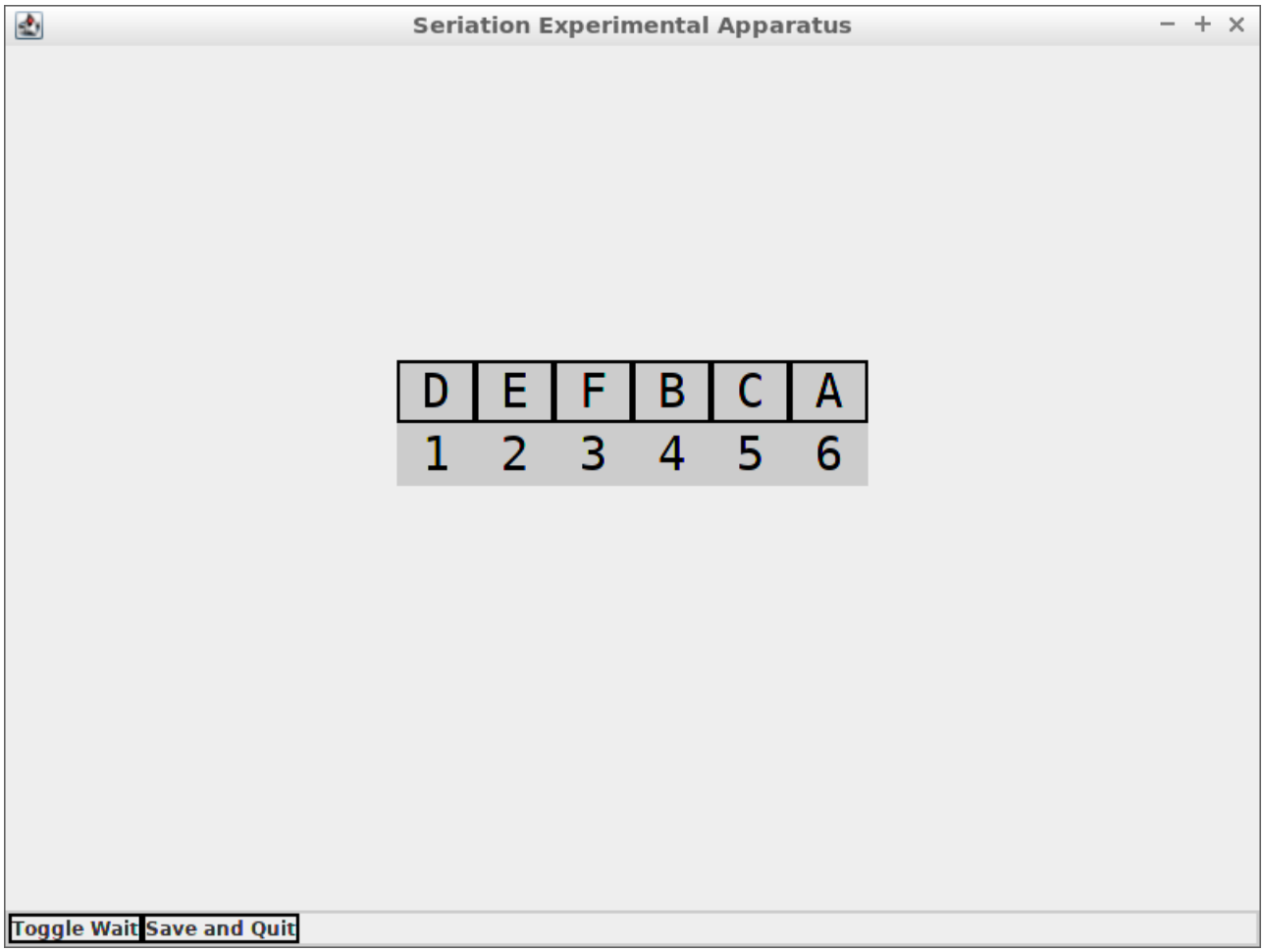

Figure 4.1: Experimental Seriation Apparatus GUI: a screen capture of the starting state of a problem, randomly chosen mid experimental run. The bottom row are slot numbers and the top row in squares are the seriation letter blocks. The Block Sorting DSL Compiler inserts an invisible Tcl/Tk version of the same interface for model testing.

To support and implement all of this, the generation of test datasets and the administration of the testing is handled by both the command line interface of the GUI, as well as the human-subject facing part of it as well.

4.1.2 Experimental Apparatus

The experiment was moderated by the use of a custom GUI, which presented a visual analog of the original seriation tasks, modified to fit the requirements of the Block Sorting Task. As can be seen in Figure 4.1, the user interface presents two rows of text on the screen. The top row are the letters of the letter blocks to be sorted, presented with black outlines around them to resemble blocks more closely. The bottom row are the slot numbers for the blocks, and corresponded with the keys on the number-row (not the numberpad) of the keyboard in their integer sequence $[1, 10]$ with 10 being mapped to the 0-key.

The two buttons in the bottom left are only used to either take a break with "Toggle Wait" or end the experiment with "Save and Quit". Subjects were instructed to not hit either of them unless instructed to do so and to hit the "Toggle Wait" button whenever they needed to take a real break or when they needed to ask a question of the administrator of the experiment. Wait times were recorded and included in the model fitting process. The exiting actually terminates the experiment and no further recording occurs (though it does include the total time including breaks from the start of the GUI until this button is struck).

While the GUI was designed to be amenable to the use of eye-tracking (by containing no extraneous visual elements), this initial work did not include recording or modeling of eye-tracking trace data. This was due in part by a desire to provide more focus on key-stroke data for the initial modeling efforts while also being influenced by

technical limitations for the eye-tracking apparatus available for use[1].

For parity, the ACT-R models are presented with a nearly identical GUI (mainly lacking only the bottom two buttons, and a change of font). The secondary GUI was integrated with ACT-R's AGI system, permitting it to be run and interacted with via the visual modules of ACT-R. This second GUI is invisible (i.e. a virtual window) by default and is only rendered on the real screen when the experimenter desires. When it does, it also can be toggled to show a circle projecting the visual attention of the ACT-R model onto the screen, like an eye-tracker.

For all intents and purposes, the models were presented with the same stimuli as the human by the experimental apparatus. The key differene is that the human needs to use the mouse to "Toggle Wait" or "Save and Exit", while the model has no mouse, and that function is handled automatically by the runtime harness, superimposing break times whenever the human being modeled took a real break, based on timers.

For all GUIs, the apparatus administers a single unsorted problem at a time, of length 4-10. While it might be possible to use longer problems with a more specialized apparatus, this limitation is based on the number rows of standard QWERTY keyboards in the US. In order to keep the focus more on algorithmic behavior rather than motor action planning, the simple arrangement was chosen.

Once a valid key is struck, the GUI will do nothing until a valid pair of numbers has been entered. Should they be a legal swap, the bottom row does not change, but the top row is instantly rearranged to show the sequence with those two positions' values moved between them. Should this swap result in a solved seriation problem, a tone would play to indicate success, and then the next problem would be displayed anew. After everything is done, and all problems are solved, another tone plays, and the apparatus automatically halts itself, just as though the "Save and Quit" button had been pressed.

4.1.3 Gathered Human Data

There were several rounds of pilot testing during which equipment was tested, and cardboard covers were added to the keyboard over the keys that the subjects were not to touch on the number pad. After some testing, it was determined that adding a similar covering over the keys besides the number row caused users to be unable to recenter their hands on the home row for touch-typists. In the end, cardboard and labeled masking tape were used to keep the subjects away from the number pad, the mouse (which were both on the desktop where they placed their hands), and the other objects nearby them in the lab (which were on shelves above and behind them). An additional cardboard partition was used to keep the subjects from looking at the recording equipment for the vision tracking system after it was determined via piloting to be very distracting.

The minimum goal for collecting data was to get roughly 90 minutes of recording time for at least 3 individual subjects. After three months, a total of 6 subjects were run through the experiment, with 4 of them providing usable data. Each subject was paid for their time, regardless of whether or not their data was used. Not all of them took the full expected 90 minutes, and not all of them took the same amount of break time either, all of which was recorded by the GUI apparatus.

Given that all of the pilot studies had determined that the attempts to salvage the eye tracker were not working, the eye tracker data was not used in any way, and no recordings were kept. However, per the IRB write-up, the appearance of calibrating and utilizing the eye tracker device was maintained for all subjects, for consistency sake.

For the subjects, one of the only qualifications for their data to be usable was whether or not they were a native English speaker (or close enough for it to be indistinguishable)–if only so that they were already familiar with the idea of the English alphabet having a clear ordering. Per the IRB, it was not permitted to ask personally identifying information such as their country of origin, but screening questions based on being able to understand some basic verbal and written instructions were permitted.

One subject failed the screening questions. Since they could not be treated differently from other subjects or ask questions outside of the IRB's approved apparatus, they were permitted to attempt the task anyway. After a few minutes of failing to grasp the instructions for the experimental activity, the subject gave up and was paid for their participation.

The other unusable subject was unable to complete the activity and had to leave after about 20 minutes of running the experiment. As their data was incomplete, they were not exposed to the full course of problems and thus would not have comparable and consistent experimental conditions to the other subjects. Their data set was discarded, after they were paid for their participation.

The full test dataset is available upon request, as are the human subject's anonymized trace datasets. All records mapping anonymized subject data to individual participants has been systematically destroyed after payments were

[1]The more than twenty year old eye-tracking equipment had already failed for an unrelated experimenter in the time between ISRB review and the administration of the experiment. For consistency, the eye-tracker was treated as though it was actually working, and subjects were kept unaware that their visual movements were not being recorded.

completed, per regulations. Anonymized paper records are retained in a locked storage area to which no one else has access. These records were written by this author, transcribing the subjects responses to entrance and exit questionnaires, so hand writing samples were not retained. The formatting is described in the Methodology chapter.

Keeping to a minimal verbal protocol, subjects were prompted with a select number of questions carefully worded to avoid introducing biases or suggesting sorting methods. Most of them were basic things about how long they thought the task took and to gauge whether or not the task was overwhelmingly tedious. The results could be summarized as it was indeed a boring thing to do, but not so much as to cause them discomfort or make them want to stop before completion. No one had an accurate estimate of how long the task had taken nor how many problems they had solved.

The only interesting feedback was that no one had noticed any of the repeaters and that most of them had directly cited using "The Alphabet Song" while solving problems. This was confirmation of what was overheard during the actual experiment, as they would mutter or sing the song while working. As well, they reported much less perceived time off-task or looking at the keyboard than was observed during the experimental process.

Taking this feedback, non-productive Operators and Alphabet Song-based Operators were added to the design of the Block Sorting DSL. As well, non-song-based Operators were retained as well, to include the possibility of some of the subjects not using the song at all, or only in part.

4.2 Detailed Contributions

This section provides a detailed summary of the novel contributions of this work.

4.2.1 Grammatical Evolution of Cognitive Models

This is the first work, which applies any form of Genetic Programming to the generation of Programs representing Cognitive Models of algorithmic behaviors. As were cited in the introduction an background section of this work, prior works have applied GP in a limited way to the generation of Expert System rulesets, as well as GA to the tuning of ACT-R control parameters (the global variables that control things like the math behind calculating recall rates, for example), but not used to generate ACT-R rule sets. These are on an entirely different level of evolutionary modeling in terms of goals, methods, as well as complexity.

The first pilot testing was done before this research agenda had fully been realized and was simply an attempt to apply GP to the automatic generation of ACT-R production rules nearly a decade ago. Several attempts all ended in failures of one kind or another, but the failures revealed that ACT-R production rules have a few properties that make them very difficult to directly evolve.

First, ACT-R models have a very strict grammar to them, and illegal programs are rejected without normally throwing an exception. Instead, designed ACT-R is generally meant for interactive or programmatic use by a modeler, who has written their code by hand, and can test it using the tooling built into ACT-R. This could include printing out detailed human-readable (but not exactly machine-parsable) diagnostic messages. These messages are ACT-R's equivalent of compiler errors, like GCC or JAVAC might produce. In addition, the grammar is embedded within Common Lisp, so it is possible to generate illegal Lisp forms as well when using a GP that is unaware of grammatical rules for its target language. Indeed, Koza's targeting of Common Lisp required all possible functions to be total and type-insensitive (as described in the Methodology section).

The second revelation was that ACT-R productions are tightly logically coupled, and almost any and every possible rule set that does not respect that will cause the exact same behavior: running briefly and producing the same noop[2] fitness behavior for the GP. From the point of view of cultivating a proper fitness landscape, this is a geometry without noticeable local curvature, where all of the landscape looks identically flat and of equally bad fitness. For such a landscape, there must be a few islands of logical coupling, where there is higher and gradiated fitness geometry. However, without some way to find these islands beside random chance, there is no feedback to help the evolutionary process escape the stepwise flatness.

The third revelation was that the sheer size of ACT-R productions was daunting from the point of view of using GP to produce them without already forcing a strict reduction in the overall complexity being generated. As a Embedded DSL within Common Lisp, ACT-R can accept any legal expression in the right place. This means that, even if one didn't need to worry about the grammatical correctness, nor the logical coupling that is required to even get ACT-R production rules to pass control through themselves in some ordering, there is still the fact that you are

[2]Noop is sometimes hyphenated No-op, NO-OP, or NOOP. It is an operation which does nothing productive, and just wastes time and computing cycles. They are sometimes done on purpose, usually to delay things while waiting on timers. Here they indicate actions which do not modify the keys being pressed, aside from perhaps causing added delays.

unconstrained in what to put in the details of the production rules. There is just too high of a dimensionality there to randomly generate.

Together, these properties and the failures leading up to them resulted in the design of using a DSL Compiler to produce only grammatically legal programs, where the GP would have a reduced Program Space complexity to explore, and where the nature of the task the DSL represented would constrain the nature of the logical flow between legal DSL Operators. For there to be an alternative to this kind of design, there would need to be some hard and computable response to the issues raised earlier. While its not impossible that such a technique might exist besides creating a DSL per task to be explored via evolution, it is highly likely that such an approach would still wind up implementing most of the same things the DSL Compiler would need to do anyway.

Now, the final system uses the Grammatical Evolution library, GEVA, to generate integer chromosomes genotypes, which are then converted using the Block Sorting Grammar BNF into Block Sorting DSL Program string phenotypes. The process by which this occurs involves each choice-point in the BNF expansion process consuming exactly one integer gene, which is $|x \bmod n|$ where x is the current integer gene, and n is the number of possible expansions of the current rule (i.e. at least 2 or else it would not be a choice point, where it is either a terminal, or a single expansion step that consumes no genes but does recurse deeper into the BNF tree). This maps all integer genes to only one of the possible expansion paths, without directly constraining the range of legal integers that gene might take on artificially. All possible genotypes are legal to produce and all should produce some kind of measurable quantitative fitness value.

The DSL handles the logical program flow, via the DSL Operators implementation of a Von Neumann Architecture (described below), and it can legally produce arbitrary complexity for the DSL programs. This is not unconstrained complexity, however, and the grammar for the DSL ensures that irrelevant computations do not intrude upon the Block Sorting Task being modeled.

This is the first work, to my knowledge that has produced something of this magnitude and comprehensively implemented it completely for an example task like the Block Sorting Task.

4.2.2 Computer-Moderated Adult Block Sorting Task

This work builds upon modeling of tasks for young children with physical objects, and turns it into a reusable computer-moderated task for adults. This novel task was created and implemented to act as an exemplar for this research agenda, to keep it from being entirely abstract. The focus is not of course on the actual results of the Block Sorting Task, but others may find it interesting in and of itself, as it is certainly based on Paigettian tasks which are of actual interest to developmental researchers.

This task is explained at some length elsewhere in this work, including in this chapter as well as the chapter on methodology. What was said there will not be repeated here. Instead, the long term prospects of the task will be outlined.

While this work does not directly model vision tracking, or fMRI BOLD Activation data traces, the Task is designed so that all three kinds of Traces can be gathered simultaneously, in theory. In fact, if such data traces were had, it should serve to very firmly provide additional feedback for fitness landscapes including during otherwise unobservable steps (e.g. keystroke traces don't inform the model about visual actions). It is perhaps ironic that adding additional Trace dimensions does not actual increase the degrees of freedom of the genotypes, and in fact only restricts the Program Space further, perhaps allowing faster elimination of low viability strategies that might otherwise be indistinguishable without the other trace data, but which would clearly be different for humans. This is because the visual actions and BOLD activations are already in the models (i.e. ACT-R produces them by default), but they are not included in the fitness calculation of Trace similarity.

Unfortunately, the degree to visual and hand movements that the task requires rules out Traces which are strongly interfered with by extraneous movements, like EEG traces. However, ACT-R does have native support for fMRI BOLD generation for it models, so it may be compared against similar human traces without significant extension.

In total, this exemplar Block Sorting Task is carefully demarked and modified from other similar seriation Tasks, and constitutes a novel and reusable portion of this work.

4.2.3 Block Sorting Grammar DSL

This work introduces the concept of not only organizing behaviors in terms of task-specific operators, but it also introduces a novel BNF Grammar to formally represent the full range of behaviors possible. This DSL representation is integrated into quantitative and executable models in this work.

Other than a select few meta-compilers like Herbal(Cohen, Ritter, & Haynes, 2010) there are very few approaches to writing models in cognitive architectures that does not just write them directly for the task being modeled. In

fact, none of them have taken the approach of restructuring the modeling process into domain-specific grammatically-constrained recombinable components, like the Operators of the Block Sorting Grammar DSL.

The technical details of this are extensively described in the methodology chapter, so they won't be repeated here. However, the design of this DSL is directly influenced by the prior novel contributions, and is unique insofar as it not only simultaneously serves as a target for evolution and automatic modeling of individuals, but also as a user-readable representation language.

With many machine learning techniques, the representation is opaque to the modeler. Whatever it is doing is all the information that the modeler can access directly. In order to avoid the opacity, this DSL approach is completely human readable DSL program code, as textual strings, and is just as readable as a Lisp program. Contrast this with the readability of Machine Learning models based on Artificial Neural Networks, where the end results are large matrices of real values weights, whose interpretation is opaque even to experts.

In fact, there is a special property in this representation that uniquely permits structural similarity metrics to compare programs automatically, to find common code snippets among them. This serves as the basis for the extended research agenda involving Strategy Groups, and being able to characterize and extract code snippets which represent real human problem solving strategies directly and quantitatively.

While the other meta-compilers like Herbal may be usable options to do some parts of this process, it is certainly not its intended use. Insofar as this is the case, this use of DSLs and reusable domain-specific Operators is unique and novel.

4.2.4 DSL Compiler

The DSL Compiler introduced in this work is new, and its method of action is likewise, representing the first of a class of special-purpose tools to reify DSL programs into cognitive architecture programs. In this work, the exemplar is the Block Sorting Grammar DSL, and the target cognitive architecture is ACT-R.

As much of the page-count of this work is spent describing this Compiler in detail, the novel place it holds is what will be examined here. While other meta-compilers may be able to represent abstract Operators, they are not presently used in that way. Instead, this Compiler is unique in that it is designed under the assertion that any good and interesting Task should have its own kinds of Operators, and its own Task-specific DSL, and DSL Compiler.

While such an assertion is perhaps controversial, it is clear that the creation of a DSL like this permits all of the same benefits for automatically modeling that Task as this exemplar Block Sorting Task has. Even without necessarily needing to evolve programs, the DSL is still a remarkably powerful modeling tool. Even non-programmers can write statements in the DSL and have them translate into working quantitative models.

This DSL Compiler is novel in its design, and in the approach to modeling that it represents, as the technical core of this work.

4.2.5 Von Neumann Architecture in ACT-R

This work creates the first detailed representation of a Von Neumann Architecture grounded in ACT-R. While this was not the original aim of the work, it was realized that nothing less powerful than a stored program computational architecture would be powerful enough to implement the whole range of possible behaviors required to represent the Block Sorting Grammar DSL, as well as the threading of control needed to execute these experiments, while also being realizable in ACT-R.

As this work represents in part the first concerted and intentional effort to make an automated way to generate all possible legal behaviors within a DSL representing a particular Task, this required using a general model of computation to string together statements of arbitrary complexity and depth. Other similar modeling efforts seem to have restricted their range of behaviors artificially, or evolved control parameters which did not actually change the complexity or depth of the behaviors, only their timings (and perhaps biases in recall or utility derived from control parameters).

This work builds upon the Von Neumann Architecture because it rather closely resembles actual cognitive behaviors involving long term memory and working memory, more closely than other potential models, like the Lambda Calculus, for example.

This is undoubtedly novel.

4.2.6 Representation of Arbitrarily Complex Nested Behavior

This is the first time that Arbitrarily Complex nested behavior is permitted in ACT-R without requiring explicit hand-coding of the complete range of programs. It was this design goal that drove the adoption of the DSL, and not

the other way around; indeed it is absolutely required for the used of Evolutionary Algorithms to fit these models.

Building upon the prior two novel contributions, this is novel.

4.2.7 Automatic Individual Modeling

This is the first time that Cognitive Models of Individuals can be automatically generated. While other automated modeling methods have existed, none have been able to create and explore the whole space of possible programs to represent an individual, only in narrower senses of fitting preexisting models. By inference, the stored program would be logically indistinguishable from a hand-coded one, as learning mechanisms would theoretically approach the same sequence of behaviors as a hand-coded one with sufficient expertise.

This contribution has been discussed indirectly earlier, but the most important point that is not included therein is that no prior attempts at Individual Modeling had an automated method for modeling individuals, merely algorithms and procedures for helping a modeler to model individuals by hand.

4.2.8 Nuanced Chunking

This work has revealed that the current Chunking behavior is not transparent enough to handle a Von Neumann Architecture, and a more nuanced representation may be required. If an amended chunking algorithm were used, the parts of the algorithmic behavior which are constant would be able to benefit from chunking, where the current chunking algorithm does not.

In practice, the ACT-R chunking algorithm discards buffer contents without consideration for also amending the rules that may have been using that buffers content, to account for it in some other way. If chunking wants to just remove it, it potentially breaks the logical flow of control. It would be better if it either found some way to leave those contents alone, or to rewrite the productions to keep them from needing those buffers when chunked, without also breaking the rules that follow the rewritten chunk. This could be thought of as an optimization to permit logical control thread tracing and production rewriting by the compiler, where certain buffers contents would be logically reducible to a series of constants in memory.

If this were working, it is possible that vast portions of a single person's behaviors which do not significantly vary during task execution to be chunked into large-scale operations. Should this work as expected, it would asymptotically approach the performance of models which were hand written to not include the overhead of the Von Neumann Architecture. This supports the similar notions from Section 4.2.7.

To my understanding, no other work has arrived at such a conclusion, in part because this is the first work implementing such a general model if computation within ACT-R.

Chapter 5

Conclusions and Future Work

5.1 Post-Compiler Research Agenda

The content of this section is tied closely to that of the Methodology Section, and it may help to refer back to it when reading the content here. An important aspect of this work is that it describes a new methodology for research into human cognitive behaviors. As such, many parts of this section discuss how to interpret the results of its application to the particular problem of sorting blocks. While understanding how people sort things is interesting, it is only an example application. Indeed the focus of this work is not specifically the mechanics of block sorting research, but on this method and how it can be applied and interpreted. Per original committee feedback, this extended agenda should was estimated as about a decade worth of work, and is discussed here to contextualize the dissertation work, and show how it leads to novel fields.

Reflecting this focus, the content here explores in abstract the process of experimenting with the method described in Section 3.1.

5.1.1 Simultaneous Evolution of Control Parameters and Coevolution of Training Sets

As has been mentioned already in earlier chapters, the immediate successor to Kase (Ritter et al., 2017) should be simultaneous evolution of ACT-R SGP control parameters alongside the evolution of ACT-R production rules. Doing so is not even hard to do, other than to amend the Block Sorting Grammar BNF to include mandatory fields that map one-to-one and onto the SGP parameters of interest. The only difficulty for such evolution is that SGP parameters may be either very sensitive or very insensitive in practice, and their ultimate effects on the fitness landscape are not entirely linear or predictable *a priori*. It may also make the search space large enough to be unwieldy as well, depending on the scope of the problem being modeled. The prerequisite for this extension is having working models which use the fixed SGP control parameters and successfully evolve normal models of humans. Only after that narrower work is proofed can the following on work be done.

Another potential avenue would be the use of Coevolution to simultaneously evolved training sets with the models that they are tested against. This is nontrivial to do in practice, given how the test subjects were not necessarily exposed to the exact same training sets in the same order. However, the real trial here would be to initially match well a particular person's behaviors, and then to use Coevolution to prove that the algorithmic behavior was not just overfitting. The Coevolutionary testing set would be evolved so as to maximize the difficulty of the model being applied against it, both of which would go through their generations simultaneously, in lockstep.

These are just some of the many possible approaches that could be taken, which are directly based on prior research, and which can be clearly grounded in current working code.

5.1.2 Experimental Tests

In this section and those that follow, a variety of testing procedures are presented for the purpose of verifying that certain properties hold for the results of this method. Since the goal of this research is to develop this as a research methodology, these tests should serve as reference points for future users of this method so that they can verify that their work also has the same properties as described here. These power testing procedures are based on normal statistical power tests, just applied to the novel case of trying to model individual behaviors instead of more typical population average behaviors.

In general, there are two levels of analysis that can be done using this method: the level that describes a single individual, and the level of the population of individuals. At each level different criteria are examined, so this work has developed distinct experiments that will verify the important criteria at each level of analysis.

While they are discussed in more detail below, the experiments generally take the form of power tests where the null hypothesis is that $H_0 : p_I < p_R$ and the alternative is $H_A : p_I \geq p_R$, where p_I is the probability that a population of interest passes some test and p_R is the probability that a reference population passes some test. The populations differ among the various tests, but they tend to be general, such as if the population of programs bred to match a particular person, or a reference population of programs bred according to some other criteria. They all have the format that the experiment requires some population of programs to be bred, and then the resulting programs are used to generate fitness values, which are compared against other fitness values resulting from members of the other population being evaluated by the same fitness function. More detailed information about this can be found in earlier sections.

Statistically speaking, the fitness values themselves are completely deterministic, but the program whose evaluation resulted in them is generated by the semi-random (heuristically guided) GP process. Since each program is a randomly chosen sample of Program Space, and the fitness measures are deterministically generated by them, then the Central Limit Theorem says that the fitnesses should follow a Normal distribution. This is helpful for analytical purposes, because few other metrics that can be gathered from Program Space follow a known statistical distribution and have such meaningful interpretations.

5.1.3 Individual-Level Tests

At the individual level, the tests compare populations bred to match specific individuals against reference populations which solve the same task as the individual, but not necessarily the same way. This begins with defining some variables of interest.

SOLVER is a population of reference solvers bred only to accomplish the task, without reference to human behaviors. Such a population may be defined Singly, consisting of a population resulting from a single GP run (which should have low diversity (Burke, Gustafson, & Kendall, 2002)), or from multiple samples of GP runs (high diversity). SOLVER may be precomputed, and a single reference population may be used instead of a new one being generated each time a comparison needs to be made.

IND_X is a population of programs bred to accomplish the task most like person X, with reference to timing and sequences. Such a population may be defined Singly, consisting of a population resulting from a single GP run (which should have low diversity), or from multiple samples of GP runs (high diversity). IND_X may be precomputed, and a single reference population may be used instead of a new one being generated each time a comparison needs to be made, but fewer comparisons are made for IND_X than SOLVER.

At this level, there are a number of forms of analysis that can be considered. Below they have been organized into three main forms of analysis, based on what is being compared, rather than the details of the comparison. Each supports hypothesis testing to show that the techniques developed here produce results with certain properties.

IND_X vs. SOLVER

To show that the process of breeding programs to match human behaviors actually contributes something noticeable to the behavior of the programs so bred, a test can be performed which compares IND_X and SOLVER. In this test, both populations are evaluated with the fitness function which assigns higher fitness to programs which generate traces that most closely match the traces of individual X. As such, the hypothesis is $H_0 : p_X < p_{\text{SOLVER}}$ and the alternative is $H_A : p_X \geq p_{\text{SOLVER}}$, where p_X is the composite percentage match of IND_X over all presented test cases, and p_{SOLVER} is the same for SOLVER. In order to do this, the top N items from each population would need to be used for the evaluation, where $1 \leq N \leq |P|$ (here P is either population). Interpreting this test is simple: if programs bred to match person X do a better job matching them that reference solutions not specifically meant to match X, then the bred programs replicate the behavior of X in a statistically significant way.

IND_X vs. IND_Y

Similar to testing against the SOLVER reference population, it is possible to compare populations derived from two different people, person X being represented by IND_X, and person Y by IND_Y. This test would consist of both populations being evaluated against the fitness function which rewards programs that match against a specific person's data (e.g. both populations would be matched against just X or Y). Here, the hypothesis would be: $H_0 : p_X < p_Y$ and the alternative is $H_A : p_X \geq p_Y$, where p_X is the composite percentage match of IND_X over all presented test cases, and p_Y is the same for IND_Y. Additionally, the converse of this test would also need to be run: $H_0 : p_X > p_Y$ and the alternative is $H_A : p_X \leq p_Y$. Interpreting this pair of tests requires

them to be seen as a way to test if those programs bred for a specific person do well only for that person, and not for someone else. Both tests should support this, or else there is difficulty statistically distinguishing X from Y using this method (indicating a need for either larger populations, or more subject data requirements).

$\bigcup_{i=1}^{n}$ IND$_{Xi}$ vs. SOLVER
In this test, finite unions of populations are compared to a single reference population. These finite unions are composed of n populations bred to match person X, such that a particular population IND$_{Xi}$ is made with a different random seed than IND$_{Xj}$ when $i \neq j$. By varying the random seed, but keeping everything else the same, we will generally get a different population with the property that it also approximates the behavior of X (specifically it should pass the first test mentioned above). When multiple populations of this sort are compared against SOLVER, their aggregate should also have the same statistical strength. Stated as a hypothesis test: $H_0 : p_X < p_{\text{SOLVER}}$ and the alternative is $H_A : p_X \geq p_{\text{SOLVER}}$, where p_X is the composite percentage match of $\bigcup_{i=1}^{n}$ IND$_i$ over all presented test cases, and p_{SOLVER} is the same for SOLVER. To interpret this test, it is to show that there are multiple ways to model a single person, and that the aggregate of these separate ways still models that person. This is semantically meaningful when the Aggregate-level tests are considered, because this is the tie-in between the levels. Failure here indicates that some unexpected behavior happens when populations are combined via union (a result which would be interesting all on its own), and that the Aggregate-level analysis may not be feasible.

This battery of Individual-level tests can be applied to the problem of block sorting, to show that the tests work much as expected. However, the real importance of these tests is that they form a consistent testing methodology such that future users of this research can apply to their own experiments to determine if they are using large enough populations, if they need more human data, or if their domain has unusual properties. This experimental methodology is much more important than the results of its application to block sorting, but it needs to be demonstrated before use.

5.1.4 Aggregated-Level Tests

After the Individual-level tests have shown that statistically significant results can be drawn from models of individual data and populations, the next step in the analysis is to show that the populations of programs do indeed form Strategy Groups[1] in Program Space. While the details of forming Strategy Groups via fuzzy clustering is detailed in Section 3.1, these tests explain whether or not these Strategy Groups are currently statistically significant on their own, or if more populations need to be added to Program Space. Unlike the Individual-level tests, these tests all assume that the Strategy Groups have already been identified by the process in Section 3.1.

There are three main forms of testing that need to be done on Strategy Groups:

Cluster Cohesion Testing
In this test, Samples drawn from a single Strategy Group are compared to each other. They are evaluated by one or more fitness functions, where their behavior should be consistently similar. These fitness functions could be ones meant to match individuals, or to simply solve the problem—it is not their absolute performance on these evaluations that matter, it is their similarity. The hypothesis test here would be $H_0 : p_{Xi} \neq p_{Xj}$ and the alternative is $H_A : p_{Xi} = p_{Xj}$, where all p terms are aggregate percentage match against test traces. The interpretation of this test is that Strategy Group members should behave similarly, regardless of how they were originated. This also ties in with the third Individual-level test described in the previous section, being another way to test the same idea. Failure here indicates that the Strategy Group is not strongly self-similar yet, and that additional populations need to be added.

Cluster Differentiation Testing
Reversing the Cluster Cohesion test above, this test compares across Strategy Groups to show that they demonstrate statistically differentiable behavior. Following the same protocol as the Cohesion test, the hypothesis becomes $H_0 : p_X = p_Y$ and the alternative is $H_A : p_X \neq p_Y$, where X and Y are Strategy Groups with little overlap (this can be formally stated in terms of membership levels and linguistic hedges). What this test means for an experimenter is that the Strategy Groups that are strongly different also produce statistically different behavior. Failure here does not have a clear meaning, unless both Strategy Groups have passed the Cohesion test—at which point it indicates something unusual about the problem being solved (e.g. topological properties of Program Space).

[1] Again, these are fuzzy clusters of common subsequences of Block Sorting DSL program's ASTs based on common code snippets representing common behavioral segments from multiple possible real DSL programs.

Cluster Power Testing

Of these tests, this is the only one that makes additional assumptions about people. If it is assumed that people can reliably find solution strategies which are heuristically superior to non-heuristic solutions, then Strategy Groups should show higher average fitnesses than programs not in Strategy Groups. This assumption is not necessarily true, but should hold statistically in many cases (it is also the least supported Power Test on this list). If this is to be tested, then members drawn randomly from Strategy Groups would be put through the same kind of test as in the first Individual-level test above, but with a fitness function which simply maximizes objective solutions. To state the hypothesis: $H_0 : p_X < p_{\text{SOLVER}}$ and the alternative is $H_A : p_X \geq p_{\text{SOLVER}}$, where p_X is the composite percentage match of IND_X over all presented test cases, and p_{SOLVER} is the same for SOLVER, and the matching is done objectively, without the individual matching. Should this test be conducted, its results could be interpreted to show support for the assumption stated above, at least for that problem. Furthermore, it could be used as a simple basis to show why individuals are a productive source for heuristics. Failure here would indicate that no strong heuristic exists, and that the humans had no real insight into solving the problem.

This Aggregate-level battery of tests complements the Individual-level, and gives insight into the properties of the problem being solved by the humans.

5.1.5 Human Data Sources

Traces are defined formally in Section 3.1.3, but they are best understood as human behavioral traces. A *Trace* in this context is a log of sequential actions, made while a person does some specific task. Sequential data is desirable for Genetic Programming training, and it is relatively common in several kinds of classical Psychology experiments. These kinds of experiments have typically been developed painstakingly by Psychologists to produce experimental protocols that reduce or eliminate many sources of noise and error in the resulting human data.

As an added benefit, some preexisting experimental data sources may be usable for research purposes (Friedrich, 2008; Friedrich & Ritter, 2020). The criteria for being usable is a fairly stringent one: each human subject must generate a series of data points which contain information about their decision made at a particular point in time, as well as the current state of the task-system. Additionally, prior experimentation conducted under Dr. Reed has established that only tasks where some kind of feedback about the efficacy of a decision is available on a per-decision basis will be usable. Other kinds of trace data lacks sufficient feedback for the EA's to use as heuristics, and results in topologies with almost perverse characteristics like severe plateauing[2]. When the wrong kind of tasks are used, the EA's run without sufficient heuristic guidance, and collapse into a Random Walk where the time to complete the process would be larger than the time before the heat-death of the universe (Page, 1976), or roughly 10^{100} years.

There are a wealth of potential tasks already available. Three types that are under investigation are:

1. Sorting Tasks (also called Seriation Tasks): these tasks consist of different kinds of sorting problems, like giving subjects variably weighted cubes and asking them to sort them in order from lightest to heaviest (Gascon, 1976). This is also the task that this research agenda is using as its exemplar task.

2. Process Control Tasks: exemplified by the Sugar Factory Task (Berry & Broadbent, 1984) in which a person is tasked with controlling a simulated Factory that makes sugar, where they need to decide each day how many workers they need the next day to meet quotas given the surpluses or shortfalls of today.

3. Temporal Sequential Decision Problems: the general term for other kinds of tasks which generate behavioral traces. Tasks that might fit here include game-play tasks, gambling tasks, and problem solving tasks (Friedrich, 2008; Friedrich & Ritter, 2020; Newell & Simon, 1972).

Regardless of which post-process tasks follow on researchers could choose, they will need to ensure that there is more than one way to solve it. Since this work is specifically interested in Strategy clustering, the kinds of tasks will need to respond to multiple Strategies. Additionally, it is desirable that the sequence of actions should be able to produce feedback about performance while you are doing them, like a sorting task should show how out-of-order it is while in the process of being sorted.

Once an experimenter has the data, they will need to do some kinds of initial clustering in order to confirm that Strategy Groups exist within the human data. Since it is not expected that the Strategies are necessarily mutually exclusive, it is intended to use Fuzzy Clustering methods. For instance, Mixed Strategies in Nash Equilibria would not be Crisp Strategies (see Section 5.3.2).

[2]These are regions in Program Space where the fitness curvature is nearly zero, giving no information to guide the heuristic.

5.1.6 Verification

There are two phases of verification that need to take place, the initial phase mentioned above, where the presence of Strategy Groups within the human data is checked, and the second phase where Strategy Group members are verified to recluster with the same groups to a strong degree.

Initial checking for Strategy Groups in the human data is planned to take place after the process has already been run once. Since all of the tasks are designed to have multiple solutions, the test for Strategy Groups is straightforward: count the number of Strategy Groups that show in the data. If that number is greater than one, then the interpretation is easy, since some variety of Strategies has shown itself in the human data. The number need not be the actual number of solutions—if there is indeed a finite set of optimal ones[3]. Rather, it should approach that number if more human data is added, or it should grow unbounded if there is no finite set of solutions for that Task.

However, if the number of clusters after process application is exactly one, then there are multiple potential causes that must be distinguished statistically. Either the humans only used a single Strategy, and the process is correctly representing that, or they used multiple Strategies which the process failed to disambiguate. The first case is easy enough to confirm by adding additional human data, which should eventually contain data representative of other Strategies, if the task is designed properly to avoid strongly biasing one or another. A failed disambiguation, however, will fail to change when more data is added. It may shift its centroid, but will only resolve to multiple Strategies if the cause of the failed disambiguation was related to insufficient data.

Secondary verification asks a different question entirely. If the Strategy Groups are actually meaningful instead of arbitrary, then program drawn from them should consistently be reclustered with the original Strategy Groups. This is more of a test of the properties of the clustering algorithms than the Strategy Groups per se, but this property is important for Strategy Groups to have. The method by which this can be tested is similar to normal statistical power testing, save that the categorization being tested is represented by multiple fuzzy sets, to which a particular data point may belong to all sets with varying degrees. Failure here would indicate a need to change clustering algorithms to something which had more stability.

5.1.7 Sampling

In order to test the Sampling part of the process described in Section 3.1.5, the Strategy Groups would be used to generate programs that should be representative of that Strategy Group, but that did not exist prior to the sample being drawn. Similarly to sampling a probability distribution, the likelihood of having a particular membership degree for a particular Strategy Group should match the statistical distribution of the extant members of that group.

To check for this, the EA process would be seeded with members of the Strategy Group in question, drawn randomly[4] from the members. The EA would run for a period of time, and some proportion of the resulting population would be taken for consideration. These programs would be checked for behavioral similarity to the members of the Strategy Group as a sanity check, and would be assigned membership degrees as if they were already members of the original Strategy Groups. When these membership values are statistically analyzed, they should be representative of the original distribution of membership values in the pre-sampling Strategy Group. If this fails, then some part of the sampling process needs to be examined more closely.

5.2 Open Issues

A number of issues remain to be resolved. Some of them are open because they depend on decisions that have not yet been made (like the choice of future tasks for other non Blocks Sorting Task human data), while others are fundamental unknowns that will not have answers until the research is done. Of the issues listed below the first two are open because future work will not have necessarily settled on the exact tasks to be done, while the rest of the subsections are all fundamental unknowns that will be resolved empirically in the course of the research. There are a few significant Open Issues that need to be addressed in the process of this research agenda, as listed below.

First, and perhaps most significantly, it will need to be verified that the basis of this later work (that human data shows clusters) is essentially sound. If it is not, none of the later work would have any hope of working, or being meaningful if it did. Once human data is available, the groups existence can be verified or not. It should be a relatively easy process, given that this work has sought out tasks which should be solvable by significantly different strategies. The human data will need to be changed into some kind of canonical format, which will then need to be clustered using some kind of Unsupervised Learning method. As has been said elsewhere, it is expected to utilize

[3] It is possible to have a non-finite number of solutions.

[4] I have yet to determine if this should be a Uniform random selection, or a fitness proportionate selection, or a membership proportionate selection, or an inverse distance from centroid proportionate selection. Empirical testing will likely show which is best.

Fuzzy Clustering Methods. If there are Clusters, they should evidence themselves by forming multiple Clusters in the data.

Secondly, it is not known if there is a strong relationship or not between the human trace data and the AST encoded structures of ACT-R programs. If the relationship is strong (which is desirable for later work), then programs which are structurally similar will create behavioral trace data that are also similar. Although this work strongly suggests that the structural similarity will have gaps in its similarity regions, it also suspects that the human trace data may not provide sufficient density for forming strategy clusters within the structure space. The first of these can be dealt with through localized search methods, to try to move the sample from one of these low fitness lacunae to a higher fitness adjacent region. The second of these is much harder to deal with, and may necessitate modification of the kinds of human data that be can utilized. Should the strong relationship prove false, and only a weak relationship evidenced by the data, then this work would need to be amended to account for this. Likely it would need to abandon the idea of Sampling altogether, and simply focus on how to overcome the weakness of the relation in the fitting process.

Third, there are several small issues related to the process of generating ACT-R programs through Genetic Programming to minimize the differences between their observed behavioral traces, and those of the human data they are to be matched against. Should it breed towards the full set of human trace data? What about to members of a strategy group? Should some kind of abstract centroid be used, or should raw human data? Similarly, past experiments have shown that there are real obstacles to getting a running program at all in ACT-R, due to its nature as a Declarative Programming Language, rather than an Imperative one. Quite simply, the behavior of the systems exhibits strong interdependencies between rules, that make them hard to develop separately and prone to thrashing. While there are ways to mitigate these and other related problems, no one to date has ever successfully applied Genetic Programming directly to ACT-R. If these issues become too much to handle, an alternative Cognitive Architecture will need to be chosen.

Finally, it is possible that any of the post-process tasks may fail. In such a case, then the researcher would have the option to either scale back, and focus on the parts that did succeed, or to focus on the negative result, and find out why it happened. The decision will be one of feasibility, since finding out why something doesn't work may not be possible, depending on the mathematical nature of the subject of study. If a proof of impossibility is possible, then it could be interesting to researchers.

5.2.1 Design of Experiments

Until a researcher has an exact task chosen and the source of the human data is determined, no detailed statistical or algorithmic analysis can be decided upon. Till that point, only rough outlines of the design are possible. This is why the Block Sorting Task was designed as an exemplar.

If a researcher is fortunate enough to find human data that has been gathered by other researchers that can be used for their use of this methodology's purposes, the design of experiments becomes easier. Depending on the details, it may be possible to reuse their experimental design, and simply apply it to the predictions made by this process, such as the Strategy Groups. If this is the case, any additional experimentation becomes limited to matching the human data with the generated data from the process.

Should it turn out that existing data sources are insufficient for this method's purposes, then actual human experimentation would be required. Of all of the tasks that have been mentioned as being possible tasks of interest, all of them have a few experimental features in common. First, none of them are legally complicated, nor would they expose a subject to any kind of harm, and they would not gather sensitive data from subjects. This means that the experiments are all safe, and relatively free of ethical encumbrances. Second, they can all be administered by a computer system under the monitoring of an experimenter. This means that none require specialized equipment in order to run the experiment (unless they choose to add addition trace types like eye tracking data or fMRI recordings). Third, all of the tasks would be used only as inputs for the process, so no special conclusions need to be drawn from the results of the experimental tasks themselves. This puts the experiments in an unusual place, where there is little analysis done upon the raw data, save that some generated data will be compared to it.

The only important thing about administering the tasks is that there needs to be stringent controls to eliminate errors not derived from the experimental subject's own decision-making process. Either a task-specific process for detecting bad data would need to be constructed, or a certain amount of noise would need to be assumed when the process is applied.

When judging the strength of the results of this body of work, there will be a need to use both statistical and algorithmic measures of the method's power. While it has been discussed elsewhere in this paper, many of the key measures are the post-process tasks discussed in Section 3.1.5. Each of these tasks includes a particular kind of verification step. Most of them amount to testing that the results of applying a particular algorithm to the input

data is statistically significant. Fortunately, the amount of data produced should be relatively large, on the order of hundreds[5]. Additionally, the majority of these category matching results collapse large number of exotic data types (tree structures, Strategy Groups, etc.) into simple category matching. Further, since the categories being matched will have fuzzy membership functions, there is the option to either use measures that work over fuzzy sets, or the researcher can defuzzify[6] the data and use more conventional statistical measures.

5.2.2 Empirical Verification of Unobservable Steps

If tasks outside of the ones that this work currently considers are to be viable, there is a need to be able to observe some kind of data regarding what the person is doing at each decision cycle of the process. Some tasks are more difficult to measure, but may be interesting for the purposes of this research. Since the DSL Compiler utilizes the ACT-R cognitive architecture, there are a few additional options that might otherwise not be obvious.

Each of these options requires expensive specialized hardware, but will produce data in parallel with the normal task recording methods. ACT-R supports theoretical models of vision as well as theoretical predictions of BOLD response levels for specific areas of the brain, based on internal activity. This means that we can utilize eye tracking systems as well as fMRI data as ways of getting insight into the internal activity of the human.

Mathematically speaking, the behavioral Trace data that has been discussed are simple time-series data. In order to accommodate these parallel sources of Trace data, the data structures that represent a Trace would need to be generalized to hold several time-series of data. Similarly the algorithms that operate on the Trace would need to be able to either pick a single Trace to examine, or be able to consider all of them at once. Not terribly complicated, though it would change the Trace-matching heuristics mentioned earlier to handle this generalization.

Given the expense and time it would take to properly utilize this kind of extra data, it would be practical to aim at not using it until absolutely necessary.

5.2.3 Sparseness of Program Space

As explained in Section 2.6, the Curse of Dimensionality is a major problem when trying to explore a high-dimensional space like Program Space. Some of the issues with it are offset by the fact that EA's do not explore all of Program Space, only the higher fitness regions are particularly well sampled. Additionally, some indications in clustering research show that having more dimensions can actually make clustering algorithms like K-Means work better.

On the other hand, the size of Program Space can cause sparseness, a condition where the shape of the fitness landscape has great distances between high fitness areas. This can exaggerate topological issues like multimodality and deceptive curvatures to cause the algorithms to take more time to find high fitness regions. More problematically this method seeks to explore the range of solutions available. If these issues all compound together, it makes exploration potentially very expensive. While some modifications to the EA's can help[7], there's no way to know if they are needed or helpful without empirical testing.

5.2.4 Durability

The idea that the problem being solved determines the landscape of Program Space was already addressed in Section 3.1.8. It is taken as an axiom by most of the other parts of this work. However, there is no data right now to support or dispute this choice of axiom.

While there might not be any evidence yet, it may become available during the course of this research agenda. Should that happen, it is possible that a test could be devised to determine if the data indicates that the landscape is not the same topology at two different points in time. A simple method would be to recalculate the fitness of several reference programs in light of new human data. If the fitnesses stay the same when more human data is added, then the Program Space would seem to be durable. Statistically significant changes would indicate otherwise, and the greater portion of this work would need revision.

5.2.5 Sampling

In addition to the simple question of whether or not a Strategy Group can be used to generate new programs that are recategorized in the same Strategy, there is another issue that requires more data to properly examine. That

[5]The size of the data produced for a single run is related to the size of the population of the EA being run, which is typically in on the order of hundreds of individuals. Not all of them need be used, but the order of hundreds is a fair approximation.

[6]Defuzzification of fuzzy data means using one of many different methods to convert the fuzzy data to crisp data.

[7]Such as island models, random seed analysis, and multiple runs, if the Program Space is durable.

issue this work refers to as Lacunae[8], which represent the potential topological problem that the space occupied by a Strategy group may be discontinuous or contain within it much lower fitness islands.

Because of Program Space being defined over Abstract Syntax Trees, with distance between two points being based on the Edit Distance of the Trees, that means that for any high-fitness program there are a number of low fitness neighbor programs. These programs can either be syntactically invalid or valid, and if valid, there is no guarantee that they will also have high fitness. This is an expected side-effect of the use of AST's to represent programs. This is the primary reason for defining Strategy Groups in terms of Fuzzy Sets rather than normal Crisp Sets. If the uncertainty about the presence or absence of Lacunae are represented in a fuzzy measure, we can still do meaningful calculations.

5.2.6 Strategic Dynamism

While there is limited support for representing Strategy changes over time by including conditional operators, it may be that actual humans have less deterministic decision processes when choosing Strategies. While the current models support awareness of problem length, there is no meta-level awareness of the prior problems that have been solved within the range of the DSL Operators.

It is an open question as to what criteria are apropos to consider, as well as how random this might process might be. If there is normal human-level randomness driving Strategy selection and transition problem-to-problem, then it may be appropriate to include additional Operators within the Block Sorting DSL. For example, a hypothetical self-awareness of inattention, stress, or intuitive complexity estimation could be the source for new Operators. As well, it is possible for the same operational principles behind the existing Von Neumann Architecture approach to stored programs could permit the creation of a secondary working memory of past problems that were solved, and if they are recalled correctly, they could be reused instead of applying the normal process for solving the problems. These Operators could be combined with the conditional operators to permit a wider range of behaviors to be represented including more self-aware Strategy changes.

5.3 Implications

Though this work would result mainly in new theoretical results and the novel modeling method (see Section 3.1), there would be a number of side benefits. The most immediate of which is the benefits for modeling Individual Differences, as detailed below. Other benefits are definitely possible, but the need for additional work in order for them to come to fruition points towards interesting directions for future research.

5.3.1 Individual Differences

Although there has been work off and on for nearly forty years on automatic modeling of individual differences, this process will be one of the first ones to bring modern techniques to bear upon it. EA's have the wonderful property of being able to find answers that its creator is unaware of, being teleonomic and evolutionary they just randomly explore solutions without forethought.

Rather uniquely for this kind of modeling, the models are explicitly partial approximations of an individual person's behaviors, and are utilized by the processes of this work without needing to be terribly accurate. While this is fine for the limited uses that they are put to in this work, it is an open question whether or not they can be used for other purposes.

If a program generated from trace data is considered as a model of the individuals thoughts, then it has a number of properties that bear consideration. First, the program is source code, and is automatically inspectable[9], so a researcher can just read the rules that the program represents. Furthermore, they could show the same to the individual, and see if any kind of feedback from the person and the rules that they are evidently using could be used to reduce error, or explain biases. Secondly, any pair of programs can be trivially compared to find where they differ, permitting people to be differentiated from one another as individuals. Third, since the programs themselves are partitioned into Strategy Groups, the Strategy of the program in question can be used to roughly categorize people into different groups. Finally, the programs can be used to improve the approximation of the individual, by using them to seed the processes from this paper in light of new data, or more time available.

Interestingly, Miwa and Simon (1993) divide the modeling of individual differences into common parts and individual parts, which map semantically onto items in my method. The common part consisting of the structurally similar items within the Strategy Group, and the individual parts being those parts which are dissimilar. Furthermore, the heuristic at the heart of each Strategy Group is mathematically the most common part of the programs involved.

[8] Named for the Latin term for holes.
[9] A trait that many Deep Learning Neural Network models cannot claim.

5.3.2 Game Theoretic Mixed Strategies

How does the Strategy Group relate to the Mixed Strategy from Game Theory? Are they equivalent or related?

Whether it is a simple analogy, or a deeper connection, there is some measure of similarity between the idea of a Mixed Strategy from Game Theory, and the Strategy Group that represents a behavioral mixture of algorithms. This similarity will become more evident when explained below.

In Game Theory, a Pure Strategy is a single choice in how to play a game, and a Mixed Strategy is an admixture of multiple Pure Strategies by behaving according to each Pure Strategy for a certain percentage of the time while playing the game. While it is outside the scope of this work to summarize decades of Game Theory research here, it is known that many games can only be played optimally or rationally by utilizing a Mixed Strategy[10].

By analogy, the Strategy Groups represent a mixture of several unobservable algorithms, making them conceptually similar to a Game Theoretic Mixed Strategy. Where a Game Theoretic Strategy is a special case of an algorithm[11], there is sufficient similarity that it might be possible to show that the Game Theoretic Mixed Strategy is a special case of a Strategy Group; a thing which often happens when comparing fuzzy algorithms with crisp ones.

What would such a relationship mean? Its not yet obvious, but it is interesting that this body of research should be able to tie into an otherwise unrelated branch of mathematics.

5.3.3 Philosophy of Mind

Given the fact that the Computational Theory of Mind (CTM) is not proven to be true, it is appropriate to consider this work as it bears upon Philosophy of Mind in general. Certainly, if CTM were held to be false, this process could be viewed as an interesting way to make approximations of human behaviors. Under this assumption, the programs found by the process hold no deeper connection than a statistical correlation with real minds. Further, the Strategy Groups would reflect only that the problems being addressed only have a certain number of solutions because of their nature, not the nature of the mind generating the behavior.

If CTM is assumed true, real interesting implications begin to emerge. First and foremost, the programs being generated in the process are approximating a particular program or program set that humans actually use. Under this assumption, it may be possible to eventually find the actual program that generated the Trace behaviors. This means that the Program Space actually has a few special members that are used by actual humans, such that the evolutionary process may be used to find them as optimal solutions to minimize the differences between the source observations and those that are generated by the process itself.

Secondly, if CTM is held to be true, then this is the first automated method for approximating or discovering the programs that are human minds. This would open entirely new horizons for researchers, and mean that the results of the process should be meaningful for psychologists and others looking for insight into specifically human minds. The limitations of this method may seem insurmountable, but with the application of new methods or mathematics, it may be possible to overcome those limits to arrive at a truly general method for approximating human minds.

Third, if CTM is held to be true, then the programs generated by this method would qualify as test subjects for further experimentation. While the accuracy of their approximation may be a parameter, the simple fact that they are representative of human minds for certain purposes is enough that they could substitute for real humans, up to a point. This could lead to cost savings as well as new forms of testing.

Finally, if CTM is held to be true, then this method is the first shot at digitizing part of a human mind. True, it only looks at a narrow slice, but if those slices could be combined with sufficiently high accuracy, CTM would say that the result would be a human. While this may sound far fetched, it is only an implication that such a thing may be possible.

If nothing else, the fact that this method strongly relates programs to mental processes may be construed as evidence towards CTM. Not that it is definitive, just that it is indicative in that direction.

5.3.4 Solicitation of Expertise

This method provides a new range of options in Expert System development. Not only can it automatically capture the experts' behavior, but it can repair, refine, and expand upon it. Additionally, the results of this process could be extracted from Program Space as heuristics derived from experts, even though the experts themselves were unable to articulate their methods.

In the initial phase of the process, Traces are used to generate a population of programs that match the greatest proportion of the Traces. This population is an initial approximation of the algorithms used by the person used to

[10] In fact, Pure Strategies are typically considered a special case of Mixed Strategies.
[11] A Decision Algorithm about how to play the game.

generate the Trace. Be it a test subject, or an Expert, the generation of the Trace captures the person solving a series of reference problems that have been specifically designed to show part of the work during the solution. Depending on the design of the reference problems, the expert's behavior of interest may be more or less represented.

While the information about a single expert is helpful, it may be incomplete, it may have failed to capture the desired behavior, or it may have captured misleading information. Normally, this would be a debilitating issue in the development of an Expert System, but this method is not only tolerant of such an error, but may even be able to correct it.

In this process, this initial population of algorithms is used to seed later stages of evolution, where the population based on the Trace of a single person is combined with the populations based on the Traces of other people. Since they are all exploring the same Program Space, the results are composable, and in being composed, tell more about where the desirable regions of Program Space are. The next round of evolution will select against poor fitness programs, regardless of which person was originally used to create them. Then, once the process is finished, and some Strategy Group has been identified, that region can be sampled to produce high fitness programs that can be used in place of the original approximations based on a single person's Trace data.

Additionally, it is possible that the evolutionary process will improve on the behavior shown by any single expert. Again using the Strategy Group, it is possible that the generated programs will be as good, or better than the ones that represented the original experts.

5.4 Conclusions

While there have been attempts to create automatic models of humans in the past, this is the first to leverage techniques from Artificial Intelligence, Machine Learning, Cognitive Science, Programming Language Theory, and Evolution all at once. Projected orthagonally to most prior works, this dissertation simultaneously explored the major ideas required to do this kind of work, while also building some concrete foundations. The biggest contribution in this author's consideration are the questions it raised; they are more fundamental than any of the tooling developed here.

Why should people care about abstract mathematical properties of the Program Space of particular tasks? Because this abstract space has deep connections to the way that people think and act, and by learning the properties of this space, we can predict and interpret human behaviors.

Hopefully, the truth of that statement should be evident by now. Given the time and resources, it is likely that this plan of research will result in insights into the inner workings of the human mind. Insights that current methods haven't even begun to approach, or even consider trying to examine. This area of research may look like a tabula rasa, a blank slate, but it is asking very old and fundamental questions with a new vocabulary. How do people think? Using the process described here, we can begin to mathematically scratch the surface of the answer.

Practical results of this work include a new formal structure for describing the range of algorithms used by human in real tasks, as well as tools permitting those structures to be usefully employed. Applications include artificial test subjects, experimental testing methods, and the automatic generation of fully descriptive models of human behavior for that task.

The most complete part of this research agenda, the exemplar Block Sorting Task and its accompanying DSL Compiler and GE infrastructure form the basic toolset for the agenda to be explored in depth in future work. This DSL Compiler proved that arbitrarily complex nestings of human behaviors require at least the computational power of a Von Neumann Architecture to be completely modeled. Tools based around evolving DSL programs to match human individuals were built around this DSL, and serve as the concrete foundations for future work.

References

A., N., Shaw, J. C., & Simon, H. A. (1958). Elements of a theory of human problem solving. *Psychological Review*, *65*(3), 151–166. 20

Anderson, J. R. (1990). *The adaptive character of thought*. Psychology Press. 14, 20

Anderson, J. R. (2009). *How can the human mind occur in the physical universe?* Oxford University Press, USA. 14, 20, 22

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*, 1036–1060. Retrieved from http://psycnet.apa.org/psycinfo/2004-19012-009 doi: 10.1037/0033-295X.111.4.1036 14, 20

Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *Proceedings of the international conference on information processing* (pp. 125–132). 14, 38

Baraldi, A., & Blonda, P. (1999a). A survey of fuzzy clustering algorithms for pattern recognition. I. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, *29*(6), 778-785. 29, 31

Baraldi, A., & Blonda, P. (1999b). A survey of fuzzy clustering algorithms for pattern recognition. II. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, *29*(6), 786-801. 29, 31

Bard, G. V. (2007). Spelling-error tolerant, order-independent pass-phrases via the Damerau-Levenshtein string-edit distance metric. In *Proceedings of the fifth australasian symposium on acsw frontiers-volume 68* (pp. 117–124). 52, 98

Bellman, R. (1961). *Adaptive control processes: A guided tour*. Princeton University Press. 31

Berry, D., & Broadbent, D. (1984). On the relationship between task performance and associated verbalizable knowledge. *The Quarterly Journal of Experimental Psychology*, *36*(2), 209-231. 112

Best, B. (2012). Inducing models of behavior from expert task performance in virtual environments. *Computational and Mathematical Organization Theory*, 1-32. Retrieved from http://dx.doi.org/10.1007/s10588-012-9136-8 doi: 10.1007/s10588-012-9136-8 23

Best, B., Fincham, J., Gluck, K., Gunzelmann, G., & Krusmark, M. (2008). Efficient use of large-scale computational resources. In *Proceedings of the seventeenth conference on behavior representation in modeling and simulation* (pp. 180–181). 23

Best, B., Furjanic, C., Gerhart, N., Fincham, J., Gluck, K., Gunzelmann, G., & Krusmark, M. (2009). *Adaptive mesh refinement for efficient exploration of cognitive architectures and cognitive models* (Tech. Rep.). DTIC Document. 23

Best, B., & Gerhart, N. (2011). An instance of what? determining cues, attributes, and goals in a virtual task environment. In *Proceedings of the 20th conference on behavior representation in modeling and simulation (BRIMS). BRIMS Society, Sundance* (pp. 58–65). 16, 23

Beyer, K., Goldstein, J., Ramakrishnan, R., & Shaft, U. (1999). When is "nearest neighbor" meaningful? *Database Theory ICDT'99*, 217–235. 29, 35

Bezdek, J. C. (1980). A convergence theorem for the fuzzy ISODATA clustering algorithms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*(1), 1–8. 30

Bezdek, J. C., Ehrlich, R., et al. (1984). FCM: the fuzzy c-means clustering algorithm. *Computers & Geosciences*, *10*(2-3), 191–203. 30

Bezdek, J. C., Hathaway, R. J., Sabin, M. J., & Tucker, W. T. (1987). Convergence theory for fuzzy c-means: Counterexamples and repairs. *IEEE Transactions on Systems, Man, and Cybernetics*, *17*(5), 873–877. 30

Bothell, D. (n.d.). ACT-R 6.0 Reference Manual [Computer software manual]. 64

Burke, E., Gustafson, S., & Kendall, G. (2002). A survey and analysis of diversity measures in genetic programming. In *Proceedings of the genetic and evolutionary computation conference* (pp. 716–723). 110

Card, S. K., Moran, T. P., & Newell, A. (1986). *The psychology of human-computer interaction*. Lawrence Erlbaum Associates. 16

Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2010). Applying software engineering to agent development. *AI Magazine*, *31*(2), 25–44. Retrieved from http://acs.ist.psu.edu/papers/cohenRHip.pdf 106

Common lisp hyperspec [Standard]. (2005). http://www.lispworks.com/documentation/HyperSpec/Body/s_progn.htm. (ANSI Common Lisp, (American National Standard X3.226); accessed 20160606) 46

Corbett, A., Koedinger, K., & Anderson, J. (1997). Intelligent tutoring systems. *Handbook of human computer interaction*, 849–874. 15

Cover, T., & Hart, P. (1967, January). Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, *13*(1), 21 –27. doi: 10.1109/TIT.1967.1053964 29

De Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)* (Vol. 75, pp. 381–392). 62

Dennett, D. C. (1992). *Consciousness explained* (1st ed.). Back Bay Books. 22, 31

Duffy, J., & Engle-Warnick, J. (1999, 24-26 June). Using symbolic regression to infer strategies from experimental data. In D. A. Belsley & C. F. Baum (Eds.), *Fifth international conference: Computing in economics and finance* (p. 150). Boston College, MA, USA. Retrieved from http://www.pitt.edu/~jduffy/docs/Usr.pdf (Book of Abstracts) 28

Ekárt, A., & Németh, S. (2000). A metric for genetic programs and fitness sharing. *Genetic Programming*, 259–270. 27

Ferreira, C. (2001). Gene expression programming: a new adaptive algorithm for solving problems. *CoRR*, *cs.AI/0102027*. 27

Ferreira, C. (2006). *Gene expression programming: Mathematical modeling by an artificial intelligence* (2nd ed.). Springer. 27

Friedrich, M. B. (2008). *Implementierung von schematischen Denkstrategien in einer höheren Programmiersprache: Erweitern und Testen der vorhandenen Resultate durch Erfassen von zusätzlichen Daten und das Erstellen von weiteren Strategien (Implementing diagrammatic reasoning strategies in a high level language: Extending and testing the existing model results by gathering additional data and creating additional strategies)* (Masters Thesis). University of Bamberg, Faculty of Information Systems and Applied Computer Science. 36, 112

Friedrich, M. B., & Ritter, F. E. (2020). Understanding strategy differences in a diagrammatic reasoning task. , *59*, 133–150. 36, 112

Friend, E. H. (1956). Sorting on Electronic Computer Systems. *J. ACM*, *3*(3), 134–168. Retrieved from http://dblp.uni-trier.de/db/journals/jacm/jacm3.html#Friend56 17

Fu, L., & Medico, E. (2007). FLAME, a novel fuzzy clustering method for the analysis of DNA microarray data. *BMC Bioinformatics*, *8*(1), 3. Retrieved from http://www.biomedcentral.com/1471-2105/8/3 doi: 10.1186/1471-2105-8-3 29

Gascon, J. (1976). *Computerized protocol analysis of the behavior of children on a weight seriation task* (Unpublished doctoral dissertation). Departement of Psychology, University of Montreal. 14, 23, 112

Gustafson, S., Burke, E., & Krasnogor, N. (2005). On improving genetic programming for symbolic regression. In *Evolutionary computation, 2005. the 2005 ieee congress on* (Vol. 1, pp. 912–919). 27

Horst, S., & Zalta, E. N. (2009, December). *The computational theory of mind* (Winter 2009 Edition ed.). Retrieved from http://plato.stanford.edu/entries/computational-mind/ 31

Information technology – Syntactic metalanguage – Extended BNF (EBNF) (Vols. ISO/IEC JTC 1/SC 22, 90.60; Standard). (1996, December). Geneva, CH: International Organization for Standardization. 38

Jones, G., & Ritter, F. (1998). Simulating development by modifying architectures. In *Proceedings of the cognitive science society* (pp. 543–548). 23

Jones, G., Ritter, F., & Wood, D. (2000). Using a cognitive architecture to examine what develops. *Psychological Science*, *11*(2), 93–100. 23

Kase, S. (2008). *Parallel genetic algorithm optimization of a cognitive model: investigating group and individual performance on a math stressor task* (Doctor of Philosophy). Pennsylvania State University College of Information Sciences and Technology. 21

Keller, J. M., Gray, M. R., & Givens, J. A. J. (1985). A fuzzy k-Nearest neighbor algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, *15*(4), 581–584. 30

Kieras, D. E., & Meyer, D. E. (1997, December). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Hum.-Comput. Interact.*, *12*(4), 391-438. Retrieved from http://dx.doi.org/10.1207/s15327051hci1204_4 doi: 10.1207/s15327051hci1204_4 19, 20

Klahr, D., Chase, W. G., & Lovelace, E. A. (1983). Structure and process in alphabetic retrieval. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *9*(3), 462. 48, 53, 54, 97

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc. 17

Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th international joint conference on artificial intelligence* (Vol. 1, p. 768-774). 24

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection* (First Printing ed.). The MIT Press. 24

Koza, J. R. (1995). Survey of genetic algorithms and genetic programming. *In Proceedings Of the WESCON 95 - Conference Record: Microelectronics, Communications Technology, Producing Quality Products, Mobile and Portable Power, Emerging Technologies*, 589—594. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.2718 24

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: an architecture for general intelligence. *Artif. Intell.*, *33*(1), 1–64. Retrieved from http://portal.acm.org/citation.cfm?id=27702 19, 20

Langley, P., Ohlsson, S., & Sage, S. (1984). A machine learning approach to student modeling. 23

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, *10*, 707-10. Retrieved from http://www.scribd.com/doc/18654513/levenshtein?secret_password=1aycnw239qw4jqjtsm34#full 30

Lloyd., S. P. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, *28*(2), 129–137. Retrieved from http://www.cs.toronto.edu/~roweis/csc2515-2006/readings/lloyd57.pdf doi: 10.1109/TIT.1982.1056489 29

Łukasiewicz, J. (1951). *Aristotle's syllogistic: from the standpoint of modern formal logic*. Clarendon Press. 50

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, *3*(4), 184–195. 41, 50

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, *63*(2), 81. 53

Miwa, K., & Simon, H. A. (1993). Production system modeling to represent individual differences: Tradeoff between simplicity and accuracy in simulation of behavior. *Prospects for artificial intelligence: Proceedings of AISB*, *93*, 158–167. 20, 22, 116

Naur, P. E. (1963). Revised report on the algorithmic language algol 60. *CACM*, *6*(1), 1–17. 14, 38

Newell, A. (1972). You can't play 20 questions with nature and win: Projective comments on the papers of this symposium. *Visual Information Processing: Proceedings*, 283. 19, 22

Newell, A. (1990). *Unified Theories of Cognition*. Cambridge: Harvard University Press. 19, 20, 22

Newell, A., & Simon, H. A. (1972). *Human problem solving*. Prentice-Hall. 20, 112

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: symbols and search. *Commun. ACM*, *19*(3), 113–126. Retrieved from http://portal.acm.org/citation.cfm?id=360022 doi: 10.1145/360018.360022 31

O'Neill, M. (2001). *Automatic programming in an arbitrary language: Evolving programs in grammatical evolution* (Unpublished doctoral dissertation). PhD thesis, University of Limerick, 2001. 24, 27

O'Neill, M., & Ryan, C. (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, *5*(4), 349-358. 14, 24, 27, 38

Page, D. N. (1976, Jan). Particle emission rates from a black hole: Massless particles from an uncharged, nonrotating hole. *Phys. Rev. D*, *13*, 198–206. Retrieved from http://link.aps.org/doi/10.1103/PhysRevD.13.198 doi: 10.1103/PhysRevD.13.198 112

Pal, N., Pal, K., Keller, J., & Bezdek, J. (2005, August). A possibilistic fuzzy c-Means clustering algorithm. *Fuzzy Systems, IEEE Transactions on*, *13*(4), 517 – 530. doi: 10.1109/TFUZZ.2004.840099 30

Pal, N. R., & Bezdek, J. C. (1995). On cluster validity for the fuzzy c-means model. *Fuzzy Systems, IEEE Transactions on*, *3*(3), 370–379. 30

Panel on Modeling Human Behavior and Command Decision Making: Representations for Military Simulations and National Research Council. (1998). *Modeling human and organizational behavior: Application to military simulations* (illustrated edition ed.). National Academies Press. 16

Peano, G. (1889). *Arithmetices principia, nova methodo exposita*. 47

Philip, B. (2005, June). A survey on tree edit distance and related problems. *Theoretical Computer Science*, *337*(1-3), 217–239. Retrieved from http://www.sciencedirect.com/science/article/pii/S0304397505000174 doi: 10.1016/j.tcs.2004.12.030 30, 31

Poli, R., Langdon, W. B., & McPhee, N. F. (2008). *A field guide to genetic programming*. Lulu Enterprises, UK Ltd. 24, 25

Rauterberg, M. (1993). AMME: an automatic mental model evaluation to analyze user behaviour traced in a finite, discrete state space. *Ergonomics*(36), 1369–1380. 20

Rauterberg, M. (1995, September). About the influence of errors and faults on human behaviour in complex systems. In *Proceedings of symposium on human interaction with complex systems in greensboro (USA).* Greensboro (USA). 20

Ritter, F. E. (1992). *A methodology and software environment for testing process model's sequential predictions with protocols* (Unpublished doctoral dissertation). Carnegie Mellon School of Computer Science, Pittsburgh, PA. 20, 124

Ritter, F. E. (1993a). Creating a prototype environment for testing process models with protocol data. In *Paper included in the Proceedings of the InterChi Research symposium.* Amsterdam: ACM CHI. 16

Ritter, F. E. (1993b). Using a cognitive architecture to add to protocol theory. In *Abstract included in the Proceedings of the III European Congress of Psychology.* Finland, July 1993. Also presented as colloquia at Queen Mary and Westfield College (U. of London), and the U. of Regensberg, Germany, July, 1993Tampare, Finland, July 1993. Also presented as colloquia at Queen Mary and Westfield College (U. of London), and the U. of Regensberg, Germany, July, 1993: Tampare. 16

Ritter, F. E., Haynes, S. R., Cohen, M., Howes, A., John, B., Best, B., ... Vera, A. (2006). High-level behavior representation languages revisited. In *Proceedings of ICCM - 2006- Seventh International Conference on Cognitive Modeling* (pp. 404–407). Trieste, Italy: Edizioni Goliardiche. Retrieved from http://acs.ist.psu.edu/papers/ritterHCHJBLJCLSAMULV06.pdf 18

Ritter, F. E., Kase, S. E., Klein, L. C., Bennet, J. M., & Schoelles, M. (2017). Fitting a model to behavior tells us what changes cognitively when under stress and with caffeine. In (pp. 109–115). Menlo Park, CA.: AAAI Press. 21, 23, 109

Ritter, F. E., & Larkin, J. H. (1994). Developing process models as summaries of HCI action sequences. *Human Computer Interaction's special issue on Exploratory Sequential Data Analysis*, *9*(3), 345–383. Retrieved from http://acs.ist.psu.edu/papers/ritterL94.pdf 16

Ritter, F. E., Tehranchi, F., Dancy, C. L., & Kase, S. E. (in press). *Some futures for cognitive modeling and architectures: Design patterns that you can too.* (Computational and Mathematical Organization Theory) 21

Ross, T. J. (2010). *Fuzzy logic with engineering applications* (3rd ed.). Wiley. 29

Ryan, C., Collins, J., & Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic programming* (pp. 83–96). Retrieved from http://dx.doi.org/10.1007/BFb0055930 24, 27

Scott, D. W. (1992). *Multivariate density estimation* (Vol. 139). Wiley Online Library. 31

Sickel, K., & Hornegger, J. (2010). Genetic programming for expert systems. In *Evolutionary computation (cec), 2010 ieee congress on* (pp. 1–8). 28

Siler, W., & Buckley, J. J. (2004). *Fuzzy expert systems and fuzzy reasoning.* Hoboken, NJ, USA: John Wiley & Sons, Inc. Retrieved from http://www3.interscience.wiley.com/cgi-bin/bookhome/109880481 29

Sleeman, D., Brown, J., et al. (1982). Intelligent tutoring systems. 15

Sun, R. (2001). *Duality of the mind: A bottom-up approach toward cognition* (1st ed.). Psychology Press. 20

Tsakonas, A., Dounias, G., Jantzen, J., Axer, H., Bjerregaard, B., & von Keyserlingk, D. (2004). Evolving rule-based systems in two medical domains using genetic programming. *Artificial Intelligence in Medicine*, *32*(3), 195–216. 28

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, *s2-42*(1), 230–265. Retrieved from http://plms.oxfordjournals.org/cgi/content/citation/s2-42/1/230 doi: 10.1112/plms/s2-42.1.230 55, 56

Von Neumann, J. (1923). Zur einführung der transfiniten zahlen. *Acta Litterarum ac Scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae, sectio scientiarum mathematicarum*, *1*, 199–208. 47

Von Neumann, J., & Godfrey, M. D. (1993). First draft of a report on the edvac. *IEEE Annals of the History of Computing*, *15*(4), 27–75. 55

Wallach, D. P., Fackert, S., & Albach, V. (2019). Predictive prototyping for real-world applications: A model-based evaluation approach based on the ACT-R cognitive architecture. In *DIS '19: Proceedings of the 2019 on Designing Interactive Systems Conference* (pp. 1495–1502). 16

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, *20*(11), 822–823. 38

Xie, X., & Beni, G. (1991). A validity measure for fuzzy clustering. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, *13*(8), 841-847. 29

Yang, M. (1993). A survey of fuzzy clustering. *Mathematical and computer modelling*, *18*(11), 1-16. 29

Young, R. M. (1976). *Seriation by Children: An Artificial Intelligence Analysis of Piagetian Task.* Birkhauser Verlag. 14, 23

Zadeh, L. (1965, June). Fuzzy sets. *Information and Control*, *8*(3), 338–353. Retrieved from http://www

.sciencedirect.com/science/article/B7MFM-4DX43MN-W3/2/f244f7a33f31015e819042700cd83047 doi: 10.1016/S0019-9958(65)90241-X 29

Zadeh, L. (1968). Fuzzy algorithms. *Information and Control*, *12*(2), 94-102. doi: 10.1016/S0019-9958(68)90211-8 29

Zadeh, L. (2008). Fuzzy logic. *Scholarpedia*, *3*(3), 1766. Retrieved from http://www.scholarpedia.org/article/Fuzzy_logic doi: 10.4249/scholarpedia.1766 29

Zadeh, L. A. (1972). A fuzzy-set-theoretic interpretation of linguistic hedges. *Cybernetics and Systems*, *2*(3), 4-34. 29

Zadeh, L. A. (1975). The concept of a linguistic variable and its application to approximate reasoning–I. *Information Sciences*, *8*(3), 199–249. Retrieved from http://www.sciencedirect.com/science/article/B6V0C-48MYP2T-4J/2/ec57b37175839e7076884f5d156fb981 doi: 10.1016/0020-0255(75)90036-5 30, 35

Zhao, C., Kaulakis, R., Morgan, J. H., Hiam, J. W., Ritter, F. E., Sanford, J., & Morgan, G. P. (2015). Building social networks out of cognitive blocks: factors of interest in agent-based socio-cognitive simulations. *Computational and Mathematical Organization Theory*, *21*(2), 115–149. 20

Zhou, C., Xiao, W., Tirpak, T. M., & Nelson, P. C. (2003). Evolving accurate and compact classification rules with gene expression programming. *Evolutionary Computation, IEEE Transactions on*, *7*(6), 519–531. 27

Zimmermann, H. (2001). *Fuzzy set theory and its applications*. Boston: Kluwer Academic Publishers. 29

Appendix A

Algorithm Appendix

A.1 Fitness Heuristics

There are a number of heuristics that been passingly mentioned:

- Trace Matching: maximize the percentage of Traces from a single human which have the highest Trace Match between the reference Traces and the Traces generated by running the programs in the EA population.
- Best-Solution: maximize some problem specific optimality criteria defined over the Traces generated by programs in the EA population.
- Strategy Centroid Distance: after taking a sample of programs near the Centroid of a Strategy Group, use them to generate Traces, which are then matched against as if the Trace Matching mentioned above used these Traces in the place of those from a human.

A.2 Trace Matching Heuristic

Data: A set of Test Condtions, C, and a set of synthetic Trace Data, S, also a set of Trace data from a specific Human for the same Conditions, H
Result: A Fitness measure, $f \in [0, 1]$, where the maximum value is preferred
matchSum $\leftarrow 0$;
count $\leftarrow 0$;
foreach $c \in C$ **do**
    synthTrace $\leftarrow$ `TraceForCondition(`c, S`)`;
    humanTrace $\leftarrow$ `TraceForCondition(`c, H`)`;
    `// LongestMatch gives the maximum percentage of overlap`
    matchSum $\leftarrow$ matchSum + `LongestMatch(`*synthTrace, humanTrace*`)`;
    count $\leftarrow$ count + 1;
end
f $\leftarrow$ matchSum / count;

Algorithm 2: Trace Matching Heursitic

A.3 Best Match Heuristic

A.4 Strategy Centroid Distance Heuristic

A.5 Trace Matching

The similarity of two Traces is relatively easy to calculate. Simply, find longest matching subsequence(Ritter, 1992), and edit distance. These two numbers are complementary, and can be used separately or aggregated as a weighted

Data: A set of Test Condtions, C, and a set of synthetic Trace Data, S also, a problem-specific optimality measure $K :: Trace \to [0, \infty)$
Result: A Fitness measure, $f \in [0, \infty)$, where the maximum value is preferred
matchSum $\leftarrow 0$;
foreach $c \in C$ **do**
  synthTrace $\leftarrow$ `TraceForCondition(`*c, S*`)`;
  traceOptimality $\leftarrow$ `Apply(`*K, synthTrace*`)`;
  matchSum $\leftarrow$ matchSum + traceOptimality;
end
f $\leftarrow$ matchSum;

Algorithm 3: Best Match Heursitic

Data: A null-root AST encoding of a Chromosome, X, and C a set of target Centroids.
Result: A Fitness measure, $f \in [0, \infty)$, where the minimum value is preferred
matchSum $\leftarrow 0$;
foreach $c \in C$ **do**
  `// EditDistance is the Rotationally Invariant algorithm below`
  dist $\leftarrow$ `EditDistance(`*X, c*`)`;
  matchSum $\leftarrow$ matchSum + dist;
end
f $\leftarrow$ matchSum;

Algorithm 4: Strategy Centroid Distance Heursitic

sum. Further, for problems with semi-numeric traces, the matching portions of the Traces can be compared using Euclidean distance measures to check for geometric similarity. Similar specialized matching criteria can be developed for most kinds of Trace data, but the details are problem specific. Instead, presented here are generic (unspecialized) Trace matching metrics. They have several important qualities: they are linear in behavior, they are normalizable, they operate on arbitrary length unmatched inputs, they are fully defined functions, and they permit biasing the importance of inputs.

In this set of equations, I describe the mathematics of Trace Matching, as a way to determine the similarity of a pair of Traces. **LCS** is the Longest Common Subsequence using simple equality, $\mathbf{LCS}_\epsilon$ is Longest Common Subsequence using an equality operator permitting a tolerance of $\pm\epsilon$ milliseconds, the α function projects a Trace into a sequence of events only without timings, the *tau* function projects a Trace into a sequence of timings only without events, the A function is the Event Similarity metric, the χ function is the Timing Similarity metric, $\|t\|$ is the length of a Trace t, and **max** is the Maximum operator, β is bias weighting (in $[0, 1]$) to increase the importance of timings or event similarities (where $\beta = 0.5$ is the unbiased weighting), and S is the composite Trace Similarity function in $[0, 1]$. Thus, the Similarity between Traces is the weighted sum of the Similarities of the Event Similarity and the Time Similarity. Both of these later metrics are the LCS divided by the longest length of the traces.

$$A(T_1, T_2) = \frac{\mathbf{LCS}(\alpha(T_1), \alpha(T_2))}{\mathbf{max}(\|T_1\|, \|T_2\|)} \tag{A.1}$$

$$\chi(T_1, T_2) = \frac{\mathbf{LCS}_\epsilon(\tau(T_1), \tau(T_2))}{\mathbf{max}(\|T_1\|, \|T_2\|)} \tag{A.2}$$

$$S(T_1, T_2, \beta) = \beta A(T_1, T_2) + (1 - \beta)\chi(T_1, T_2) \tag{A.3}$$

In order to generalize this definition of similarity to multidimensional Traces—those which feature multiple independent input channels, such as driving a car while navigating—a slight redefinition needs to occur. First, each let us define a c-Trace as one with c independent input channels, where each channel is a normal Trace. Each individual channel is numbered from 0 to $c - 1$, with their ordering being arbitrary but consistent, so Trace n channel i is written T_{ni}. Further, each channel i has its own bias operator β_i (which follows the rules for the bias operator above). Finally, each channel is given a generalize contribution weighting ω_i such that $\omega_i \in [0, 1]$ and $\sum_{i=o}^{c-1} \omega_i = 1$. A unbiased weighting is $\omega_i = \frac{1}{c}$. Then the Generalized Similarity for c-Traces of higher dimensions than $c = 1$ is:

$$S_c(T_1, T_2, \bar{\beta}, \bar{\omega}) = \sum_{i=0}^{c-1} \omega_i \left(\beta_i A(T_{1i}, T_{2i}) + (1 - \beta_i)\chi(T_{1i}, T_{2i})\right) \tag{A.4}$$

With unbiased weightings, this becomes[1]:

$$S_c(T_1, T_2) = \frac{1}{2c}\left(\sum_{i=0}^{c-1} A(T_{1i}, T_{2i}) + \sum_{i=0}^{c-1} \chi(T_{1i}, T_{2i})\right) \tag{A.5}$$

The reader may confirm that these functions fulfill the criteria described above as being good candidate Trace Matching metrics. They are linear, normalized, variably biased, total functions that operate over arbitrary length inputs. As such, they can be used as the basis for the Trace Matching Fitness Function defined in Appendix A.1 above.

A.6 Specialized Tree-Edit Distance

Distance measure based on rotationally invariant edit-distance. This is the edit distance of according to the following rule: the tree has a null root, and each subtree is an IF-THEN rule; for each pair of trees to be compared, take the sum of the edit distances of the IF-parts and THEN-parts, where each potential pair of IFs and each potential pair of THENs get compared, and only those parts that are closest count, and if multiple ones are closest, then choose one at random. This represents the idea that the ordering of the rules doesn't matter, so only the closest ones matter.

Data: A pair of trees, A and B, describing a null-rooted set of IF-THEN subtrees
Result: An edit distance, $\delta \in \mathbb{Z}^*$
$\delta \leftarrow 0$;
`// Walk through all pairs of IF-THENs.`
foreach *subtree* $i \in A$ **do**
  condA $\leftarrow$ `IfPart(`i`)`;
  exprA $\leftarrow$ `ThenPart(`i`)`;
  diffA $\leftarrow \infty$;
  foreach *subtree* $j \in B$ **do**
    condB $\leftarrow$ `IfPart(`j`)`;
    exprB $\leftarrow$ `ThenPart(`j`)`;
    `// EditDistance always returns a finite value <` ∞
    diffABCond $\leftarrow$ `EditDistance(`*condA, condB*`)`;
    diffABExpr $\leftarrow$ `EditDistance(`*exprA, exprB*`)`;
    diffA $\leftarrow$ `Min(`*diffABCond + diffABExpr, diffA*`)`;
  end
  `// Prevents empty trees from causing errors.`
  if *diffA* $< \infty$ **then**
    $\delta \leftarrow \delta +$ diffA;
  end
end

Algorithm 5: Rotationally Invariant Tree-Edit Distance

A.7 Mapping Fitness to Virtual Membership

Another interesting phenomenon: fitness values can be transformed into membership values. Given a fitness function which maximizes optimality as it approaches infinity, its values can be mapped one-to-one onto real values between zero and one, such as those used for membership functions.

Formally, fitness values can be transformed into membership values by a function f:

$$f :: [0, \infty) \to [0, 1] \tag{A.6}$$

$$f(0) = 0 \tag{A.7}$$

[1]By breaking the summation into smaller summations, the calculations may be done independently and in parallel. Which is efficient for HPC usage. This is not to say that the biased equation is inefficient, just that the unbiased one has better theoretical speedup under Amdahl's Law.

$$\lim_{x \to \infty} f(x) = 1 \tag{A.8}$$

$$f(x) = \frac{1}{1 + e^{k-x}} - \frac{1}{1 + e^{k}} \text{ where } k \in [0, \infty) \tag{A.9}$$

Here, Equation (A.9) is an example mapping equation[2]. Since the mapping is non-unique, any suitable function can be used. The one here is a variable sigmoidal curve, where k can be used to control the steepness of the curve.

While it's application may not be readily apparent, the majority of Fuzzy Set and Fuzzy Logic operations require some membership function to be computed. Since so much of the key components of the method use such operators, it may at some point be desirable to map the fitness of a program from its native type to the restricted $[0, 1]$ range that is more common. For example, to compute an analog of the fuzzy set: high fitness members of a particular Strategy Group; the fitness of the programs would be turned into membership values in $[0, 1]$ and membership in the Strategy Group would also be in $[0, 1]$, so when the t-norm analog of the Boolean **AND** operator is applied, the arguments work normally, and the result is interpretable in light of preexisting fuzzy logic literature. Additionally the same method could be used to make distances between programs into the same domain of $[0, 1]$.

[2]Thanks to J. Nicholas Hobbs for this example equation.

Appendix B

Code Appendix

This Appendix contains code excerpts that were too long[1] to fit within the main text of this work.

B.1 Compiler-Sequence-For

```
(defun compiler-sequence-for (base-symbol base-operator-str-name arity)
  (if (zerop arity)
    ;;handle 0-arity behavior
    (progn
     #'(lambda
      (compiler-state
       args
       parent-sym my-sym
       return-sym return-state return-op
       parent parent-arg-number)
      (destructuring-bind
       (thisid)
       (register-idents
        (list base-symbol :pop)
        compiler-state)
             (list
              (make-instance
               'dsl-op-sequence
               :name  base-operator-str-name
               :branch-name  thisid
               :branch-order 0
               :done? t
               :arity 0
               :args  nil
               :return-branch   return-sym
               :return-state    return-state
               :return-operator return-op
               :parent          parent
               :parent-argument-number parent-arg-number)))))
    ;;else handle Arity >= 1
    (progn
     #'(lambda
        (compiler-state
         args
         parent-sym my-sym
         return-sym return-state return-op
         parent parent-arg-number)
        (let* ((arglen (length args))
               (this-dsl-op-obj-str-name (dsl-symbol-to-string base-symbol))
               (this-dsl-op-obj (gethash this-dsl-op-obj-str-name
                                         *dsl-operators*))
               (this-dsl-op-obj-jump-fn
                (lambda (step argnum)
                 (if argnum
                  (jump-state-lookup this-dsl-op-obj-str-name step argnum)
                  (jump-state-lookup this-dsl-op-obj-str-name step))))
```

[1] Per committee feedback, any code listing longer than two double-sided printed pages would be too long.

```
(thisid-ident1
 (car
   (register-idents
    (list base-symbol)
      compiler-state)))
(this-dm-id-override-fn
  (lambda (n)
   (dsl-string-to-symbol
     (string-upcase
      (format nil
       (if (numberp n)
         "~a-dm~d"
         "~a-dm-~a")
       (dsl-symbol-to-string thisid-ident1)
       n)))))
(arg-literals (mapcar #'(lambda (x) (not (listp x))) args))
(non-literal-count (count nil arg-literals))
(all-literals? (zerop non-literal-count))
(dm-args (mapcar #'(lambda (a b) (if a b :empty)) arg-literals args))
(last-dm-args (mapcar #'(lambda (x) :empty) args))
(first-dm (make-instance
           'dsl-op-sequence
           :name  base-operator-str-name
           :branch-name  thisid-ident1
           :branch-order 0
           :arity arity
           :args  dm-args
           :done? nil ; all-literals?
           :return-branch parent
           :return-state  return-state
           :return-operator return-op
           :parent parent
           :parent-argument-number parent-arg-number
           :dm-identity-override (funcall this-dm-id-override-fn 0)))
(last-dm (make-instance ;;last element all done!
          'dsl-op-sequence
          :name  base-operator-str-name
          :branch-name  thisid-ident1
          :branch-order arglen
          :arity arity
          :args  last-dm-args
          :done? t
          :return-branch parent
          :return-state  return-state
          :return-operator return-op
          :parent parent
          :parent-argument-number parent-arg-number
          :dm-identity-override (funcall this-dm-id-override-fn 'return)))
(rest-dm (remove nil
          (loop for i from 1 below arglen
           collecting
           (let* ((x (elt args i))
                  (literal? (not (listp x))))
            (let* ((thisargid thisid-ident1)
                   (literal-args literal?)
                    (regular-args dm-args))
              (make-instance
               'dsl-op-sequence
               :name  base-operator-str-name
               :branch-name  thisargid
               :branch-order i
               :arity arity
               :args  dm-args
               :args-literal-value literal-args
               :done? :empty
               :return-branch parent
               :return-state  return-state ;here
               :return-operator return-op
               :parent parent
               :parent-argument-number parent-arg-number
               :dm-identity-override
                (funcall this-dm-id-override-fn i)))))))
```

```
(temp-ordering
  (cons first-dm (append rest-dm (list last-dm))))
(temp-ordering-length (length temp-ordering))
(temp-order-len (1- (length temp-ordering)))
;;Notes:
;; 0. The following number of DM elements should be
;;   generated:
;;    0-arity: 1
;;    N-arity: N+1 , with branch-orders [0,N]
;; 1. a DM element should never have an empty return-*
;;    section (literals just have the next step in
;;    the current operator to jump to)
;; 2. a DM element without a literal value in it
;;    should contain a jump value in return-* under
;;    the following cases:
;;    a. The last element in the DM contains the
;;       address of the parent
;;       branch/state/operator pair that the
;;       computation should jump to when completed
;;    b. If you are not the last element, then the
;;       jump info should point to the
;;       branch/state/operator pair that the
;;       computation should jump to when ready to
;;       continue
(res
 (loop for i from 0 below temp-ordering-length
  appending
   (progn
    (if (= i (1- temp-ordering-length))
     (list (elt temp-ordering i))
     (let* ((this-operator (elt temp-ordering i))
            (this-dm-jump-state-number (funcall this-dsl-op-obj-jump-fn "return" i))
            (this-arg (elt args i))
            (am-i-a-literal? (elt arg-literals i))
            (my-child-dms
             (if (not am-i-a-literal?)
               (let*
                ((subexpression-for-this-arg this-arg)
                 (subexpr-operator-name (first subexpression-for-this-arg))
                 (subexpr-operator-name-str (dsl-symbol-to-string subexpr-operator-name))
                 (subexpr-args (rest subexpression-for-this-arg))
                 (dsl-operator-for-subexpr (gethash subexpr-operator-name-str *dsl-operators*))
                 (compiler-fn (compiler-for dsl-operator-for-subexpr))
                 (child-sym   (register-idents (list (dsl-string-to-symbol (name
                 ↪  dsl-operator-for-subexpr)))
                                                                          compiler-state))
                 (child-dm-elements
                  (funcall compiler-fn
                   compiler-state
                   subexpr-args
                   (branch-name this-operator) ;parent-sym
                   child-sym ;my-sym
                   (dsl-string-to-symbol (branch-name this-operator)) ;return-sym
                   this-dm-jump-state-number ;return state
                   (dsl-string-to-symbol (name this-operator)) ;return-op
                   (branch-name this-operator) ;parent
                   i          ;parent-arg-number
                   )))
                (pop-ident compiler-state)
                child-dm-elements)
               nil))
            (next-operator
             (if am-i-a-literal?
              (progn
                 (elt temp-ordering (1+ i)))
              (first my-child-dms))))
       (let* ((y this-operator)
              (x next-operator))
        (setf (return-branch y) (branch-name x))
        (setf (return-state  y) (branch-order  x))
        (setf (return-operator y) (dsl-string-to-symbol (name x)))
        ;;return this list to the
```

```
                    ;;appendding operator in loop
                    (append (list this-operator) my-child-dms))

                                    ))
                                ;;recurse on arg i's
                                ;;subexpression, unless you
                                ;;are a literal

                                ;;literal: set jump to
                                ;;coordinates for next part
                                ;;of operator

                                ;;subexpr: get dm-element for
                                ;;entry to subexpression,
                                ;;jump to it
                                )

                          )
                    ;;setf return branch, state, operators for
                    ;;first to point at the last

                    ;; reverse the list from this:
                    ;; (loop for i from 0 up to length of list
                    ;;       (reverse (cons first-dm rest-dm))
                    ;;    do (setf (... return branch, operator,
                    ;;       state) to the  info for the next
                    ;; one in the list
                    )
                   )
              res)))))
```

Listing 30: Compiler-Sequence-For Listing

B.2 Main Loop Production Rules

```
(p main-start-timer
        =goal>
        ISA metaproc
        current-branch :empty
        branch-order   :empty
        subgoal :start-timer
        ?temporal>
        state free
        ==>
        !eval! (setf *last-time-we-solved-a-problem* (mp-time-ms)
                     )
        +temporal>
        ISA time
        =goal>
        subgoal :empty
        )

     (p main-check
        =goal>
        ISA metaproc
        current-branch :empty
        branch-order   :empty
        subgoal :empty
        < loop-iteration ,*max-iterations*
        loop-iteration =iteration
        dm-reload :empty
        ?retrieval>
        state free
        - state error
        ==>
        +retrieval>
        ISA op-sequence
        branch-name  ,*first-branch-id*
        branch-order 0
```

```
   =goal>
   current-branch ,*first-branch-id*
   branch-order 0
   )

(p main-retrievenext ;;should only be called when returning
   ;;from a call to a subexpression
   =goal>
   ISA metaproc
   - current-branch :empty
   current-branch ,*first-branch-id*
   - branch-order   :empty  ;=border
   branch-order   =border
   - dm-reload  :empty
   dm-reload    =reload
   < loop-iteration ,*max-iterations*
   loop-iteration =iteration
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  ,*first-branch-id*
   branch-order =border
   =goal>
   current-branch ,*first-branch-id*
   branch-order =border
   subgoal :empty
   dm-reload :empty
   )

(p main-root-next-step
   =goal>
   ISA metaproc
   current-branch ,*first-branch-id*
   branch-order =border
   < branch-order   ,*main-loop-cutoff-length*
   subgoal :empty
   dm-reload :empty
   loop-iteration =iteration
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name   ,*first-branch-id*
   branch-order  =step-number
   done          =done
   return-branch =return-branch
   return-state  =return-state
   return-operator =return-op
   op-name       =op
   arg0          =arg0
   arg1          =arg1
   arg2          =arg2
   arg3          =arg3
   arg4          =arg4
   arg5          =arg5
   arg6          =arg6
   ==>
   =goal>
   current-branch     =return-branch
   branch-order =return-state
   return-value    :empty
   operator       =return-op
   arg0          =arg0
   arg1          =arg1
   arg2          =arg2
   arg3          =arg3
   arg4          =arg4
   arg5          =arg5
```

```
   arg6          =arg6

   )

(p main-next-loop
   =goal>
   ISA metaproc
   current-branch ,*first-branch-id*
   current-problem =current-problem
   length          =length
   ;;> branch-order 0
   branch-order   ,*main-loop-cutoff-length*
   branch-order =border
   subgoal :empty
   dm-reload :empty
   loop-iteration =iteration
   time-since-process-start =timestart
   time-since-last-break    =timebreak
   time-current             =timecurr
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name   ,*first-branch-id*
   branch-order  =step-number
   done          t
   return-branch =return-branch
   return-state  =return-state
   op-name       =op
   arg0          =arg0
   arg1          =arg1
   arg2          =arg2
   arg3          =arg3
   arg4          =arg4
   arg5          =arg5
   arg6          =arg6
   =temporal>
   ISA time
   ticks =ticks
   ?temporal>
   state free
   ==>
   =temporal>
   !bind! =next-loop-num (1+ =iteration)
   !bind! =cummulative   =ticks
   !bind! =cummulativebrk (- =ticks =timebreak)
   !bind! =cummtimecurr  (- =ticks =timecurr)
   !eval! (setf *accumulated-edit-distance*
   (+ *accumulated-edit-distance*
      (string-diff =current-problem (alpha-subseq =length t))))
   =goal>
   loop-iteration =next-loop-num
   current-branch     :empty
   subgoal            :empty
   branch-order       :empty
   return-value    :empty
   operator       :empty
   arg0          =arg0
   arg1          =arg1
   arg2          =arg2
   arg3          =arg3
   arg4          =arg4
   arg5          =arg5
   arg6          =arg6
   dm-reload     :empty
   time-since-process-start =cummulative
   time-since-last-break    =cummulativebrk
   time-current             =cummtimecurr

   )
```

```
;detect correctly done, reset for next problem, signalled by
;tone being played and asynchronously placed in the
;audio-location buffer until attended to,


(p main-next-problem
    =goal>
    ISA metaproc
    current-problem =old-current
    starting-order  =old-starting
           time-since-process-start =timestart
    time-since-last-break    =timebreak
    time-current             =timecurr
    ?aural-location>
    state    free  ; always t
    attended nil   ; has not been sent to AURAL buffer
    finished t     ; tone done playing (700ms)
    =aural-location>
    ISA audio-event
    kind     =kind
    location =location
    onset    =onset
    ?aural>
    state free
    =temporal>
    ISA time
    ticks =ticks
    ?temporal>
    state free
    ==>
    !bind! =nextbreak
      (if *break-times* (car (car *break-times*)) nil)
    !bind! =triggerbreak
      (if (and =nextbreak (>= =ticks (/ =nextbreak 1000))) t nil)
    !bind! =newsubgoal (if =triggerbreak :take-break :empty)
    !bind! =newdmreload
      (if =triggerbreak (+ =ticks (/ (car (cdr (car *break-times*))) 1000)))
    !bind! =new-current *current-problem*
    !bind! =new-length  (length *current-problem*)
    !bind! =new-time (mp-time-ms)
    !bind! =new-starting (copy-seq =new-current)
    !bind! =cummulative   =ticks
    !bind! =cummulativebrk (if =triggerbreak 0 (+  =ticks =timebreak))
    !eval! (setf *last-time-we-solved-a-problem* (mp-time-ms))
    !eval! (setf *accumulated-edit-distance* 0)
    ;bind the new info for the once-only stuff
    +aural>
    ISA sound
    event =aural-location
    =temporal>
    =goal>
    current-problem    =new-current
    starting-order     =new-starting
    last-problem       =old-starting
    length             =new-length
    loop-iteration     0
    current-branch     :empty
    subgoal            =newsubgoal
    branch-order       :empty
    return-value    :empty
    operator       :empty
    arg0          :empty
    arg1          :empty
    arg2          :empty
    arg3          :empty
    arg4          :empty
    arg5          :empty
    arg6          :empty
    dm-reload     =newdmreload
    timestamp     =new-time
    time-since-process-start =cummulative
    time-since-last-break    =cummulativebrk
```

```
   time-current             0
   )

(p main-wait-break
   =goal>
   ISA metaproc
   current-problem =old-current
   starting-order  =old-starting
          time-since-process-start =timestart
   time-since-last-break    =timebreak
   time-current             =timecurr
   subgoal :take-break
   dm-reload =target-time
   < dm-reload =ticks
   ?aural-location>
   state    free  ; always t
   attended nil   ; has not been sent to AURAL buffer
   finished t     ; tone done playing (700ms)
   =temporal>
   ISA time
   ticks =ticks
   ?temporal>
   state free
   ==>
   !eval! (setf *is-currently-break-time* t)
   =temporal>
   =goal>
   )

(p main-break-done
   =goal>
   ISA metaproc
   current-problem =old-current
   starting-order  =old-starting
          time-since-process-start =timestart
   time-since-last-break    =timebreak
   time-current             =timecurr
   subgoal :take-break
   dm-reload =target-time
   >= dm-reload =ticks
   ?aural-location>
   state    free  ; always t
   attended nil   ; has not been sent to AURAL buffer
   finished t     ; tone done playing (700ms)
   =temporal>
   ISA time
   ticks =ticks
   ?temporal>
   state free
   ==>
   !eval! (when *break-times*  (pop *break-times*))
   !eval! (setf *is-currently-break-time* nil)
   !eval! (setf *last-time-we-solved-a-problem* (mp-time-ms))
   =temporal>
   =goal>
   subgoal :empty
   dm-reload :empty
   )

(p main-problem-exhausted
   =goal>
   ISA metaproc
   >= loop-iteration ,*max-iterations*
   ==>
   !eval! (experiment-halt nil)
   !stop!

   )
```

Listing 31: Main Loop Production Rules

B.3 READ-WHOLE Operator

```
(define-operator
    :name "read-whole"
  :arity 0
  :compiler-for (compiler-sequence-for 'read-whole "read-whole" 0)
  :prod-jumps   '(entry-point
                  lookat
                  return-from)
  :dm-jumps     '(dm-recall-parent)
  :productions  `(
                  (p read-whole-seq
                    =goal>
                    ISA metaproc
                    current-branch =branch
                    branch-order   0
                    operator      read-whole
                    length        =len
                    ==>
                    =goal>
                    branch-order ,(jump-state-lookup "read-whole" 'lookat)
                    subgoal      0
                    dm-reload    :empty
                    length       =len
                    arg0         0
                    arg1         0
                    )

                    ,@(screen-pos-literal-generator
                      #'(lambda (len x y x-index y-index)
                          (let* ((x-lower (- x +X-LOWER-TOLERANCE+))
                                 (x-upper (+ x +X-UPPER-TOLERANCE+))
                                 (y-lower (- y +Y-LOWER-TOLERANCE+))
                                 (y-upper (+ y +Y-UPPER-TOLERANCE+)))

                            `(p ,(screen-pos-literal-name-generator
                                  "read-whole" len x-index y-index)
                                =goal>
                                ISA metaproc
                                current-branch     =branch
                                branch-order       ,(jump-state-lookup "read-whole" 'lookat)
                                operator           read-whole
                                subgoal            ,x-index
                                dm-reload          :empty
                                length             ,len
                                arg0               ,x-index
                                arg1               ,y-index
                                ?visual-location>
                                state free
                                ==>
                                +visual-location>
                                ISA visual-location
                                >  screen-x ,x-lower
                                <= screen-x ,x-upper
                                >  screen-y ,y-lower
                                <= screen-y ,y-upper
                                kind text
                                =goal>
                                subgoal     ,(if (>= (1+ x-index) len) :lookat-done (1+ x-index) )
                                arg0        ,(1+ x-index)
                                dm-reload   ,(if (> (1+ x-index) len) :empty :attend)
                                )
                             )


                          )
                       )


                  (p read-whole-lookat-dolook
                     =goal>
                     ISA metaproc
```

```
   current-branch     =branch
   branch-order       ,(jump-state-lookup "read-whole" 'lookat)
   operator           read-whole
   subgoal            =x-index
   dm-reload          :attend
   length             =len
   arg0               =arg0
   arg1               =arg1
   =visual-location>
   ISA visual-location
   screen-x =screenx
   screen-y =screeny
   kind text
   ?visual-location>
   state free
   ?visual>
   state free
   ==>
   +visual>
   ISA move-attention
   screen-pos =visual-location
   =goal>
   dm-reload :encode
   )

(p read-whole-lookat-encode
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "read-whole" 'lookat)
   operator           read-whole
   subgoal            =x-index
   dm-reload          :encode
   length             =len
   arg0               =x
   arg1               =y
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?imaginal>
   state free
   ==>
   +imaginal>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   =visual>
   =goal>
   dm-reload :cleanup
   )

(p read-whole-lookat-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "read-whole" 'lookat)
   operator           read-whole
   subgoal            =x-index
   dm-reload          :cleanup
   length             =len
   arg0               =arg0
   arg1               =arg1
   ?imaginal>
   state free
   ==>
   -imaginal>
   =goal>
   dm-reload          :rehersal
```

```
   )

(p read-whole-lookat-rehersal
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "read-whole" 'lookat)
   operator           read-whole
   subgoal            =x-index
   dm-reload          :rehersal
   length             =len
   arg0               =x
   arg1               =y
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   =visual>
   =goal>
   dm-reload :rehersal-cleanup
   )

(p read-whole-lookat-rehersal-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "read-whole" 'lookat)
   operator           read-whole
   subgoal            =x-index
   dm-reload          :rehersal-cleanup
   length             =len
   arg0               =arg0
   arg1               =arg1
   =retrieval>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   ;?imaginal>
   ;state free
   ?retrieval>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   dm-reload          :empty
   )


(p read-whole-lookat-done
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "read-whole" 'lookat)
   operator           read-whole
   subgoal            :lookat-done
   dm-reload          :empty
```

```
      length              =len
      arg0                =arg0
      arg1                =arg1
      ?imaginal>
      state free
      ==>
      -imaginal>
      =goal>
      branch-order        ,(jump-state-lookup "read-whole" 'return-from)
      subgoal             :empty
      dm-reload           :empty
      )

   (p read-whole-return-load
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order    ,(jump-state-lookup "read-whole" 'return-from)
      operator        read-whole
      subgoal         :empty
      ?retrieval>
      state free
      - state error
      ==>
      +retrieval>
      ISA op-sequence
      branch-name     =branch
      branch-order    ,(jump-state-lookup "read-whole" 'dm-recall-parent)
      op-name         read-whole
      =goal>
      subgoal         :load
      )

   (p read-whole-return
       =goal>
       ISA metaproc
       current-branch =branch
       branch-order    ,(jump-state-lookup "read-whole" 'return-from)
       operator        read-whole
       subgoal         :load
       =retrieval>
       ISA op-sequence
       branch-name     =branch
       branch-order    ,(jump-state-lookup "read-whole" 'dm-recall-parent)
       op-name         read-whole
       return-branch   =return-branch
       return-state    =return-state
       return-operator =return-op
       ?retrieval>
       state free
       - state error
       ==>
       =goal>
       current-branch =return-branch
       branch-order    =return-state
       operator        =return-op
       return-value    :no-value
       subgoal         :empty
       arg0            :empty
       arg1            :empty
       dm-reload       :reload
       next-branch   :empty
       next-branch-number     :empty
       next-operator :empty
       )

   )
)
```

Listing 32: READ-WHOLE Operator Listing

B.4 SHIFT-HAND Operator

```
(define-operator
    :name "shift-hand"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from
                     do-operator
                     really-return
                     )
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    )
    :compiler-for (compiler-sequence-for 'shift-hand "shift-hand" 1)
    :productions
    `(
      (p shift-hand
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator       shift-hand
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "shift-hand" 'dm-recall 0)
         op-name      shift-hand
         =goal>
         branch-order ,(jump-state-lookup "shift-hand" 'recall 0)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'shift-hand "shift-hand" 1 0 :type-number)

      (p shift-hand-arg0-done
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "shift-hand" 'done 0)
         operator       shift-hand
         starting-order =starting-order
         loop-iteration =loop-iteration
         timestamp      =timestamp
         ?retrieval>
         state free
         - state error
         =retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
         op-name      shift-hand
         return-branch    =return-branch
         return-state     =return-state
         return-operator  =return-op
         timestamp      =timestamp
         loop-iteration =loop-iteration
         last-argument  0
         problem        =starting-order
         ?imaginal>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
```

```
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   - arg0       :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "shift-hand" 'return-from)
   operator           shift-hand
   dm-reload :reload
   )

(p shift-hand-return-right0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "shift-hand" 'return-from)
   operator       shift-hand
   return-value   0
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   - arg0       :empty
   arg0         =arg0
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "shift-hand" 'do-operator)
   operator           shift-hand
   subgoal            :right
   dm-reload          :empty
   )
(p shift-hand-return-right
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "shift-hand" 'return-from)
   operator       shift-hand
   >= return-value      6
   ?imaginal>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "shift-hand" 'do-operator)
   operator           shift-hand
   subgoal            :right
   dm-reload          :empty
   )

(p shift-hand-return-left
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "shift-hand" 'return-from)
```

```
   operator       shift-hand
   > return-value       0
   < return-value       6
   ?imaginal>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   - arg0       :empty
   arg0         =arg0
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "shift-hand" 'do-operator)
   operator           shift-hand
   subgoal            :left
   dm-reload          :empty
   )

(p shift-hand-find-start-fail
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "shift-hand" 'return-from)
   operator       shift-hand
   return-value   :no-value
   subgoal        :empty
   ?imaginal>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   - arg0       :empty
   arg0         =arg0
          ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "shift-hand" 'really-return)
   operator           shift-hand
   )



  ,@(motor-keystroke-literal-generator
  #'(lambda (key-horizontal key-vertical)
      ;;remember to pick the hand based on the number 1-5 L, 0 + 6-9 R
      `(
        (p ,(motor-keystroke-literal-name-generator "shift-hand-0-ply" key-horizontal key-vertical "left")
           =goal>
           ISA metaproc
           current-branch     =branch
           branch-order       ,(jump-state-lookup "shift-hand" 'do-operator)
           operator           shift-hand
           subgoal            :left
           dm-reload          :empty
           length             =len
           return-value       =key0
           return-value       ,key-horizontal
```

```
   ?retrieval>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?manual>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   +manual>
   ISA point-hand-at-key
   hand left
   to-key  =key0
   =goal>
   branch-order ,(jump-state-lookup "shift-hand" 'really-return)
   subgoal :empty
   dm-reload :empty
   )

(p ,(motor-keystroke-literal-name-generator "shift-hand-1-ply" key-horizontal key-vertical "right")
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "shift-hand" 'do-operator)
   operator           shift-hand
   subgoal            :right
   dm-reload          :empty
   length             =len
   return-value       =key0
   return-value       ,key-horizontal
   ?retrieval>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?manual>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
   op-name      shift-hand
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   +manual>
   ISA point-hand-at-key
   hand right
```

```
                to-key  =key0
                =goal>
                branch-order ,(jump-state-lookup "shift-hand" 'really-return)
                subgoal      :empty
                dm-reload    :empty
                )
              )
            )
        )

      (p shift-hand-really-return
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "shift-hand" 'really-return)
         operator       shift-hand
         - return-value :empty
         return-value   =return-value
         =retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order  ,(jump-state-lookup "shift-hand" 'dm-recall-parent)
         op-name      shift-hand
         done         t
         arg0         :empty
         arg1         :empty
         arg2         :empty
         return-branch    =return-branch
         return-state     =return-state
         return-operator  =return-op
         ?manual>
         state free
         ?retrieval>
         state free
         - state error
         ==>
         ,@(log-return-value "shift-hand" :no-value)
         =goal>
         current-branch     =return-branch
         branch-order       =return-state
         operator           =return-op
         arg0               :empty
         return-value       :no-value
         dm-reload :reload
         subgoal   :empty
         )

      )
    )
```

Listing 33: SHIFT-HAND Operator Listing

B.5 LOOK-OFF-SCREEN Operator

```
(define-operator
    :name "look-off-screen"
    :arity 1
    :compiler-for (compiler-sequence-for 'look-off-screen "look-off-screen" 1)
    :prod-jumps  '(entry-point
                   recall-arg0
                   jump-arg0
                   return-arg0
                   save-arg0
                   done-arg0
                   lookup-coords
                   return-from)
    :dm-jumps    '(dm-recall-arg0
                   dm-recall-parent
                   )
```

```
:productions
`(
  (p look-off-screen
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   0
     operator      look-off-screen
     ?retrieval>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "look-off-screen" 'dm-recall 0)
     op-name      look-off-screen
     =goal>
     branch-order ,(jump-state-lookup "look-off-screen" 'recall 0)
     subgoal      :empty
     )

  ,@(argument-p-sequence-for 'look-off-screen "look-off-screen" 1 0 :type-number)

  (p look-off-screen-arg0-done
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "look-off-screen" 'done 0)
     operator       look-off-screen
     length         =length
     starting-order =starting-order
     loop-iteration =loop-iteration
     timestamp      =timestamp
     =retrieval>
     ISA op-sequence
     branch-name    =branch
     branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
     op-name        look-off-screen
     return-branch  =return-branch
     return-state   =reutrn-state
     return-operator =return-op
     ?retrieval>
     state free
     - state error
     ?imaginal>
     state free
     ==>
     !bind!       =arg0 (act-r-random =length)
     !bind!       =arg1 (act-r-random 2)
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
     op-name      look-off-screen
     - arg0           :empty
     timestamp      =timestamp
     loop-iteration =loop-iteration
     last-argument  0
     problem        =starting-order
     =goal>
     branch-order ,(jump-state-lookup "look-off-screen" 'lookup-coords)
     arg0         =arg0
     arg1         =arg1
     subgoal      :prepare
     dm-reload    :empty
   )

  (p look-off-screen-prep-coords
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "look-off-screen" 'lookup-coords)
```

```
   operator       look-off-screen
   subgoal       :prepare
   dm-reload     :empty
   - return-value  :empty
   - return-value  :no-value
   return-value  =x
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
   op-name        look-off-screen
   ?retrieval>
   state free
   - state error
   ?temporal>
   state free
   =temporal>
   ISA time
   ticks =ticks
   ?visual>
   state free
   ==>
   !bind! =xint (if (numberp =x) =x (or (parse-integer =x :junk-allowed t) 1))
   !bind! =time (+ =ticks =xint)
   =temporal>
   =retrieval>
   +visual>
   ISA clear
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
   operator           look-off-screen
   subgoal            :wait
   dm-reload          :empty
   arg2               =time
   ;next-branch  =return-branch
   ;next-branch-number   =return-state
   ;next-operator =return-op
   )

(p look-off-screen-prep-coords-bad-no0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "look-off-screen" 'lookup-coords)
   operator       look-off-screen
   return-value  :no-value
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
   op-name        look-off-screen
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ?temporal>
   state free
   =temporal>
   ISA time
   ticks =ticks
   ?visual>
   state free
   ==>
   !bind! =time (+ =ticks 0)
   =temporal>
   =retrieval>
   +visual>
   ISA clear
```

```
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
   operator           look-off-screen
   subgoal            :wait
   dm-reload          :empty
   arg2               =time
   )

(p look-off-screen-prep-coords-wait-notyet
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
   operator           look-off-screen
   subgoal            :wait
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   arg2               =time
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
   op-name        look-off-screen
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?temporal>
   state free
   =temporal>
   ISA time
   ticks =ticks
   < ticks =time
   ?visual>
   state free
   ==>
   =temporal>
   =retrieval>
   =goal>
   subgoal :wait
   )

(p look-off-screen-prep-coords-wait-ready
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
   operator           look-off-screen
   subgoal            :wait
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   arg2               =time
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
   op-name        look-off-screen
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?temporal>
   state free
   =temporal>
   ISA time
   ticks =ticks
   >= ticks =time
   ?visual>
   state free
```

```
    ==>
    =temporal>
    =retrieval>
    =goal>
    subgoal :match
    )

,@(screen-pos-literal-generator
    #'(lambda (len x y x-index y-index)
        (let* ((x-lower (- x +X-LOWER-TOLERANCE+))
               (x-upper (+ x +X-UPPER-TOLERANCE+))
               (y-lower (- y +Y-LOWER-TOLERANCE+))
               (y-upper (+ y +Y-UPPER-TOLERANCE+)))

          `(p ,(screen-pos-literal-name-generator
                "look-off-screen" len x-index y-index)
              =goal>
              ISA metaproc
              current-branch     =branch
              branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
              operator           look-off-screen
              subgoal            :match
              dm-reload          :empty
              length             ,len
              arg0               ,x-index
              arg1               ,y-index
              =retrieval>
              ISA op-sequence
              branch-name    =branch
              branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
              op-name        look-off-screen
              return-branch  =return-branch
              return-state   =return-state
              return-operator =return-op
              ?visual-location>
              state free
              ==>
              =retrieval>
              +visual-location>
              ISA visual-location
              >  screen-x ,x-lower
              <= screen-x ,x-upper
              >  screen-y ,y-lower
              <= screen-y ,y-upper
              kind text
              =goal>
              subgoal     :lookat
              )
          )
        )
    )

 (p look-off-screen-lookat
    =goal>
    ISA metaproc
    current-branch     =branch
    branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
    operator           look-off-screen
    subgoal            :lookat
    dm-reload          :empty
    length             =len
    arg0               =x
    arg1               =y
    =visual-location>
    ISA visual-location
    screen-x =screenx
    screen-y =screeny
    kind text
    ?visual-location>
    state free
    ?visual>
    state free
```

```
   ==>
   +visual>
   ISA move-attention
   screen-pos =visual-location
   =goal>
   subgoal :encode
   )

(p look-off-screen-encode  ;;This represents unintentional sightings, so don't apply rehersal to it
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
   operator           look-off-screen
   subgoal            :encode
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?imaginal>
   state free
   ==>
   +imaginal>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   =visual>
   =goal>
   subgoal :cleanup
   )

(p look-off-screen-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-off-screen" 'lookup-coords)
   operator           look-off-screen
   subgoal            :cleanup
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   -imaginal>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
   op-name        look-off-screen
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   branch-order ,(jump-state-lookup "look-off-screen" 'return-from)
   subgoal :empty
   )
```

```
      (p look-off-screen-return
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "look-off-screen" 'return-from)
         operator        look-off-screen
         =retrieval>
         ISA op-sequence
         branch-name    =branch
         branch-order   ,(jump-state-lookup "look-off-screen" 'dm-recall-parent)
         op-name        look-off-screen
         return-branch  =return-branch
         return-state   =return-state
         return-operator =return-op
         done          t
         arg0          :empty
         arg1          :empty
         arg2          :empty
         ?retrieval>
         state free
         ==>
         =goal>
         current-branch =return-branch
         branch-order   =return-state
         operator       =return-op
         arg0           :empty
         arg1           :empty
         arg2           :empty
         return-value   :no-value
         dm-reload      :reload
         next-branch  :empty
         next-branch-number    :empty
         next-operator :empty
         subgoal   :empty
         )

      )

    )
```

Listing 34: LOOK-OFF-SCREEN Operator Listing

B.6 ONCE-ONLY Operator

```
(define-operator
    :name "once-only"
    :arity 1
    :prod-jumps    '(entry-point  ;;reserve 0, don't use it for
                     ;;anything here
                     recall-past
                     ;past-decide ;;probably don't need this
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from)
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    dm-recall-past
                    )
    :compiler-for (compiler-sequence-for 'once-only "once-only" 1)
    :productions
    `(
      (p once-only
         =goal>
         ISA metaproc
         current-branch =branch
```

```
   branch-order   0
   operator       once-only
   current-problem =problem
   timestamp       =time
   starting-order  =starting
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "once-only" 'dm-recall-past)
   op-name      once-only
   arg0         :no-value
   =goal>
   branch-order ,(jump-state-lookup "once-only" 'recall-past)
   return-value :no-value
   )

(p once-only-recall-past-success
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-only" 'recall-past)
   operator       once-only
   timestamp      =time
   starting-order  =starting
   current-problem =problem
   return-value         =oldreturn
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "once-only" 'dm-recall-past)
   op-name      once-only
   arg0         :no-value
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "once-only" 'dm-recall-past)
   op-name      once-only
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   return-value       :no-value
   subgoal            :wait
   )

(p once-only-recall-past-success-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-only" 'recall-past)
   operator       once-only
   timestamp      =time
   starting-order  =starting
   current-problem =problem
   return-value    =value
   subgoal        :wait
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
```

```
   branch-name  =branch
   branch-order ,(jump-state-lookup "once-only" 'dm-recall-past)
   op-name      once-only
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   ;arg0         =value
   ;arg1         =time
   ;arg2         =starting
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   =goal>
   current-branch     =return-branch
   branch-order       =return-state
   operator           =return-op
   return-value       =value
   arg0               :empty
   dm-reload          :reload
   subgoal            :empty
   )

(p once-only-recall-past-failure
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-only" 'recall-past)
   operator       once-only
   timestamp      =time
   return-value   =oldreturn
   ?retrieval>
   state free
   state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "once-only" 'dm-recall 0)
   op-name      once-only
   =goal>
   branch-order ,(jump-state-lookup "once-only" 'recall 0)
   subgoal      :empty
   )


,@(argument-p-sequence-for 'once-only "once-only" 1 0 :type-letter)

(p once-only-arg0-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-only" 'done 0)
   operator       once-only
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-only" 'dm-recall-parent)
   op-name          once-only
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
```

```
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-only" 'dm-recall-parent)
   op-name      once-only
   - arg0       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  0
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "once-only" 'return-from)
   operator           once-only
   subgoal   :encode
   dm-reload :reload
   )

(p once-only-return-encode
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-only" 'return-from)
   operator       once-only
   subgoal        :encode
   current-problem =problem
   starting-order  =starting
   timestamp       =time
   ?imaginal>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-only" 'dm-recall-parent)
   op-name      once-only
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-only" 'dm-recall-parent)
   op-name          once-only
   done             t
   arg0             :empty
   arg1             :empty
   arg2             :empty
   ;arg0             =arg0
   ;arg1             =time
   ;arg2             =starting
   +imaginal>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-only" 'dm-recall-past)
   op-name          once-only
   arg0             :no-value
   arg1             :empty
   arg2             :empty
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   =goal>
   subgoal :commit
   )

 (p once-only-return-commit
```

```
        =goal>
        ISA metaproc
        current-branch =branch
        branch-order   ,(jump-state-lookup "once-only" 'return-from)
        operator       once-only
        subgoal        :commit
        current-problem =problem
        starting-order  =starting
        timestamp       =time
        return-value    =value
        ?imaginal>
        state free
        =imaginal>
        ISA op-sequence
        branch-name  =branch
        branch-order  ,(jump-state-lookup "once-only" 'dm-recall-past)
        op-name          once-only
        arg0             :no-value
        ?retrieval>
        state free
        - state error
        =retrieval>
        ISA op-sequence
        branch-name  =branch
        branch-order  ,(jump-state-lookup "once-only" 'dm-recall-parent)
        op-name      once-only
        done             t
        arg0             :empty
        arg1             :empty
        arg2             :empty
        return-branch    =return-branch
        return-state     =return-state
        return-operator  =return-op
        ==>
        -imaginal>
        =goal>
        current-branch     =return-branch
        branch-order       =return-state
        operator           =return-op
        subgoal            :empty
        dm-reload          :reload
        )


      )
    )
```

Listing 35: ONCE-ONLY Operator Listing

B.7 SWAP Operator

```
(define-operator
    :name "swap"
  :arity 2
  :compiler-for (compiler-sequence-for 'swap "swap" 2)
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 lookup-coords
                 prep0
                 press0
```

```
               prep1
               press1
               return-from)
:dm-jumps    '(dm-recall-arg0
               dm-recall-arg1
               dm-recall-parent
               )
:productions
`(
  (p swap
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   0
     operator      swap
     ?retrieval>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "swap" 'dm-recall 0)
     op-name      swap
     =goal>
     branch-order ,(jump-state-lookup "swap" 'recall 0)
     subgoal      :empty
     dm-reload    :empty
     )

  ;;load data from arg0 -> X
  ,@(argument-p-sequence-for 'swap "swap" 2 0 :type-number)

  (p swap-arg0-done
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "swap" 'done 0)
     operator       swap
     subgoal      :empty
     dm-reload    :empty
     starting-order =starting-order
     loop-iteration =loop-iteration
     timestamp      =timestamp
     =retrieval>
     ISA op-sequence
     branch-name    =branch
     branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
     op-name        swap
     return-branch  =return-branch
     return-state   =reutrn-state
     return-operator =return-op
     ?retrieval>
     state free
     - state error
     ?imaginal>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "swap" 'dm-recall 1)
     op-name      swap
     =goal>
     branch-order ,(jump-state-lookup "swap" 'recall 1)
     subgoal      :empty
     dm-reload    :empty
     )

  ;;load data from arg1 -> Y

  ,@(argument-p-sequence-for 'swap "swap" 2 1 :type-number)
```

```
(p swap-arg1-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'done 1)
   operator       swap
   subgoal      :empty
   dm-reload    :empty
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name      swap
   done         t
   - arg0       :empty
   - arg1       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'lookup-coords)
   operator           swap
   subgoal            :prepare
   dm-reload          :empty
   )

;;load data from stored results into RETRIEVAL (match on fields)
(p swap-prep-coords
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'lookup-coords)
   operator       swap
   subgoal       :prepare
   dm-reload     :empty
   length        =length
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   >= arg0        0
   <=  arg0        =length
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   >= arg1        0
   <=  arg1        =length
```

```
   arg1          =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p swap-prep-coords-bad-no0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'lookup-coords)
   operator       swap
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0         :no-value
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (incf *bad-swap-params-count*)
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0 )
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   arg0               1
   arg1               1
   )

(p swap-prep-coords-bad-no1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'lookup-coords)
   operator       swap
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
```

```
   arg1         :no-value
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (incf *bad-swap-params-count*)
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0 )
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   arg0               1
   arg1               1
   )

(p swap-prep-coords-bad-low0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'lookup-coords)
   operator       swap
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   < arg0         0
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (incf *bad-swap-params-count*)
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   arg0               1
   arg1               1
   )

(p swap-prep-coords-bad-low1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'lookup-coords)
   operator       swap
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   < arg1         0
   ?imaginal>
```

```
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (incf *bad-swap-params-count*)
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   arg0               1
   arg1               1
   )

(p swap-prep-coords-bad-high0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'lookup-coords)
   operator       swap
   subgoal       :prepare
   dm-reload     :empty
   length        =length
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   > arg0         =length
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (incf *bad-swap-params-count*)
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   arg0               1
   arg1               1
   )

(p swap-prep-coords-bad-high1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'lookup-coords)
   operator       swap
   subgoal       :prepare
   dm-reload     :empty
   length        =length
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   > arg1         =length
   ?imaginal>
```

```
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (incf *bad-swap-params-count*)
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   arg0               1
   arg1               1
   )

;;prep 0; do 0; wait 0; prep 1; do 1; wait 1; return
(p swap-press0-prep-zero
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   length             =len
   arg0               0
   arg1               =key1
   ==>
   =goal>
   branch-order ,(jump-state-lookup "swap" 'press0)
   subgoal :right
   )

(p swap-press0-prep-left
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   length             =len
   > arg0             0
   < arg0             6
   arg0               =key0
   arg1               =key1
   ==>
   =goal>
   branch-order ,(jump-state-lookup "swap" 'press0)
   subgoal :left
   )

(p swap-press0-prep-right
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep0)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   length             =len
   >= arg0             6
   arg0               =key0
   arg1               =key1
   ==>
   =goal>
   branch-order ,(jump-state-lookup "swap" 'press0)
   subgoal :right
   )
```

```
,@(motor-keystroke-literal-generator
    #'(lambda (key-horizontal key-vertical)
        ;;remember to pick the hand based on the number 1-5 L, 0 + 6-9 R
        ;;make one for press0 and another for press1, with correct jumps back for each
        `( ;we want to match arg0=x y=2, and then arg1=x y=2
          (p ,(motor-keystroke-literal-name-generator "swap0-ply" key-horizontal key-vertical "left")
             =goal>
             ISA metaproc
             current-branch     =branch
             branch-order       ,(jump-state-lookup "swap" 'press0)
             operator           swap
             subgoal            :left
             dm-reload          :empty
             length             =len
             arg0               =key0
             arg0               ,key-horizontal
             arg1               =key1
             ;arg1               ,key-vertical
             ?manual>
             state free
             ==>
             +manual>
             ISA point-hand-at-key
             hand left
             to-key  =key0
             =goal>
             branch-order ,(jump-state-lookup "swap" 'press0)
             subgoal      :left
             dm-reload    :press
             )

          (p ,(motor-keystroke-literal-name-generator "swap0-punch" key-horizontal key-vertical "left")
             =goal>
             ISA metaproc
             current-branch     =branch
             branch-order       ,(jump-state-lookup "swap" 'press0)
             operator           swap
             subgoal            :left
             dm-reload          :press
             length             =len
             arg0               =key0
             arg0               ,key-horizontal
             arg1               =key1
             ;arg1               ,key-vertical
             ?manual>
             state free
             ==>
             +manual>
             ISA    punch
             hand   left
             finger index
             =goal>
             branch-order ,(jump-state-lookup "swap" 'prep1)
             subgoal      :empty
             dm-reload    :empty
             )

          (p ,(motor-keystroke-literal-name-generator "swap0-ply" key-horizontal key-vertical "right")
             =goal>
             ISA metaproc
             current-branch     =branch
             branch-order       ,(jump-state-lookup "swap" 'press0)
             operator           swap
             subgoal            :right
             dm-reload          :empty
             length             =len
             arg0               =key0
             arg0               ,key-horizontal
             arg1               =key1
             ?manual>
             state free
             ==>
```

```
            +manual>
            ISA point-hand-at-key
            hand right
            to-key  =key0
            =goal>
            branch-order ,(jump-state-lookup "swap" 'press0)
            subgoal      :right
            dm-reload    :press
            )
          (p ,(motor-keystroke-literal-name-generator "swap0-punch" key-horizontal key-vertical "right")
            =goal>
            ISA metaproc
            current-branch     =branch
            branch-order       ,(jump-state-lookup "swap" 'press0)
            operator           swap
            subgoal            :right
            dm-reload          :press
            length             =len
            arg0               =key0
            arg0               ,key-horizontal
            arg1               =key1
            ;arg1               ,key-vertical
            ?manual>
            state free
            ==>
            +manual>
            ISA    punch
            hand   right
            finger index
            =goal>
            branch-order ,(jump-state-lookup "swap" 'prep1)
            subgoal      :empty
            dm-reload    :empty
            )

          )

        )
    ;; these are the last args to the motor-generator
    nil   ;arg1-not-actr-vertical (vertical should always be row 2)
    nil   ;start-at-zero (horizontal, key y=2 x=0 is tilde, skip it 1=1 9=9 10=0-key)
    )

(p swap-press1-prep-zero
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep1)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   length             =len
   arg1               0
   arg1               =key1
   ==>
   =goal>
   branch-order ,(jump-state-lookup "swap" 'press1)
   subgoal :right
   )

(p swap-press1-prep-left
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep1)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   length             =len
   > arg1             0
   < arg1             6
   arg0               =key0
```

```
   arg1              =key1
   ==>
   =goal>
   branch-order ,(jump-state-lookup "swap" 'press1)
   subgoal :left
   )

(p swap-press1-prep-right
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "swap" 'prep1)
   operator           swap
   subgoal            :empty
   dm-reload          :empty
   length             =len
   >= arg1             6
   arg0               =key0
   arg1               =key1
   ==>
   =goal>
   branch-order ,(jump-state-lookup "swap" 'press1)
   subgoal :right
   )

,@(motor-keystroke-literal-generator
    #'(lambda (key-horizontal key-vertical)
        ;;remember to pick the hand based on the number 1-5 L, 0 + 6-9 R
        ;;make one for press0 and another for press1, with correct jumps back for each
        `(
          (p ,(motor-keystroke-literal-name-generator "swap1-ply" key-horizontal key-vertical "left")
             =goal>
             ISA metaproc
             current-branch     =branch
             branch-order       ,(jump-state-lookup "swap" 'press1)
             operator           swap
             subgoal            :left
             dm-reload          :empty
             length             =len
             arg0               =key0
             arg1               =key1
             arg1               ,key-horizontal
             ?manual>
             state free
             ==>
             +manual>
             ISA point-hand-at-key
             hand left
             to-key  =key1
             =goal>
             branch-order ,(jump-state-lookup "swap" 'press1)
             subgoal      :left
             dm-reload    :press
             )

          (p ,(motor-keystroke-literal-name-generator "swap1-punch" key-horizontal key-vertical "left")
             =goal>
             ISA metaproc
             current-branch     =branch
             branch-order       ,(jump-state-lookup "swap" 'press1)
             operator           swap
             subgoal            :left
             dm-reload          :press
             length             =len
             arg0               =key0
             arg1               =key1
             arg1               ,key-horizontal
             ?manual>
             state free
             ==>
             +manual>
             ISA    punch
```

```
                hand   left
                finger index
                =goal>
                branch-order ,(jump-state-lookup "swap" 'return-from)
                subgoal      :empty
                dm-reload    :empty
                )

             (p ,(motor-keystroke-literal-name-generator "swap1-ply" key-horizontal key-vertical "right")
                =goal>
                ISA metaproc
                current-branch     =branch
                branch-order       ,(jump-state-lookup "swap" 'press1)
                operator           swap
                subgoal            :right
                dm-reload          :empty
                length             =len
                arg0               =key0
                arg1               =key1
                arg1               ,key-horizontal
                ?manual>
                state free
                ==>
                +manual>
                ISA point-hand-at-key
                hand right
                to-key  =key1
                =goal>
                branch-order ,(jump-state-lookup "swap" 'press1)
                subgoal      :right
                dm-reload    :press
                )
             (p ,(motor-keystroke-literal-name-generator "swap1-punch" key-horizontal key-vertical "right")
                =goal>
                ISA metaproc
                current-branch     =branch
                branch-order       ,(jump-state-lookup "swap" 'press1)
                operator           swap
                subgoal            :right
                dm-reload          :press
                length             =len
                arg0               =key0
                arg1               =key1
                arg1               ,key-horizontal
                ?manual>
                state free
                ==>
                +manual>
                ISA    punch
                hand   right
                finger index
                =goal>
                branch-order ,(jump-state-lookup "swap" 'return-from)
                subgoal      :empty
                dm-reload    :empty
                )

             )

           )
         ;; these are the last args to the motor-generator
         nil   ;arg1-not-actr-vertical (vertical should always be row 2)
         nil   ;start-at-zero (horizontal, key y=2 x=0 is tilde, skip it 1=1 9=9 10=0-key)
         )

(p swap-note-changes
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :empty
```

```
   dm-reload      :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           =arg0
   - arg1         :empty
   arg1           =arg1
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   =goal>
   subgoal     :arg0-letters-recall
   arg0   =arg0
   arg1   =arg1
   )

(p swap-arg0-letters-recall
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-letters-recall
   arg0           =arg0
   arg1           =arg1
   =retrieval>
   ISA letters
   slot-number    =arg0
   row-number     0
   letter         =textvalue
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   =goal>
   subgoal     :arg1-letters-recall
   arg2        =textvalue
   )

(p swap-arg1-letters-recall
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-letters-recall
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   =retrieval>
   ISA letters
   slot-number    =arg1
   row-number     0
   letter         =textvalue
   ?retrieval>
   state free
   - state error
```

```
   ?imaginal>
   state free
   - state error
   ==>
   +imaginal>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   -retrieval>
   =goal>
   subgoal     :arg0-commit-change
   arg3        =textvalue
   )

(p swap-arg0-commit-change
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-commit-change
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -imaginal>
   =goal>
   subgoal     :arg0-reherse-0

   )

(p swap-arg0-reherse-0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-0
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   =goal>
   subgoal     :arg0-reherse-recall-0

   )

  (p swap-arg0-reherse-recall-0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
```

```
   operator        swap
   subgoal         :arg0-reherse-recall-0
   arg0            =arg0
   arg1            =arg1
   arg2            =arg0text
   arg3            =arg1text
   =retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg0-reherse-cleanup-0

   )

(p swap-arg0-reherse-cleanup-0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal         :arg0-reherse-cleanup-0
   arg0            =arg0
   arg1            =arg1
   arg2            =arg0text
   arg3            =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg0-reherse-1

   )

;;do this 3 times here
(p swap-arg0-reherse-1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal         :arg0-reherse-1
   arg0            =arg0
   arg1            =arg1
   arg2            =arg0text
   arg3            =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==> ;;positions and text are swapped
   +retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
```

```
   =goal>
   subgoal     :arg0-reherse-recall-1
   )

(p swap-arg0-reherse-recall-1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-recall-1
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   =retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg0-reherse-cleanup-1
   )

(p swap-arg0-reherse-cleanup-1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-cleanup-1
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg0-reherse-2
   )

(p swap-arg0-reherse-2
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-2
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
```

```
   ==>
   +retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   =goal>
   subgoal     :arg0-reherse-recall-2

   )

(p swap-arg0-reherse-recall-2
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-recall-2
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   =retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg0-reherse-cleanup-2
   )

(p swap-arg0-reherse-cleanup-2
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-cleanup-2
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg0-reherse-3
   )

(p swap-arg0-reherse-3
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-3
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
```

```
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   =goal>
   subgoal     :arg0-reherse-recall-3
   )

(p swap-arg0-reherse-recall-3
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-recall-3
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   =retrieval>
   ISA letters
   slot-number =arg1
   row-number  0
   letter      =arg0text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg0-reherse-cleanup-3
   )

(p swap-arg0-reherse-cleanup-3
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg0-reherse-cleanup-3
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-letters-store

   )

(p swap-arg1-letters-store
   =goal>
   ISA metaproc
```

```
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-letters-store
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   +imaginal>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   -retrieval>
   =goal>
   subgoal     :arg1-commit-change
   )

(p swap-arg1-commit-change
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-commit-change
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -imaginal>
   =goal>
   subgoal     :arg1-reherse-0
   )

(p swap-arg1-reherse-0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-0
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   =goal>
```

```
   subgoal     :arg1-reherse-recall-0
   )

(p swap-arg1-reherse-recall-0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-recall-0
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   =retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-reherse-cleanup-0
   )

(p swap-arg1-reherse-cleanup-0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-cleanup-0
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-reherse-1
   )

(p swap-arg1-reherse-1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-1
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
```

```
   +retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   =goal>
   subgoal     :arg1-reherse-recall-1
   )

(p swap-arg1-reherse-recall-1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-recall-1
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   =retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-reherse-cleanup-1
   )

(p swap-arg1-reherse-cleanup-1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-cleanup-1
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-reherse-2
   )

(p swap-arg1-reherse-2
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-2
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
```

```
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   =goal>
   subgoal     :arg1-reherse-recall-2
   )

(p swap-arg1-reherse-recall-2
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-recall-2
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   =retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-reherse-cleanup-2
   )

(p swap-arg1-reherse-cleanup-2
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-cleanup-2
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-reherse-3
   )

(p swap-arg1-reherse-3
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
```

```
   subgoal        :arg1-reherse-3
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   =goal>
   subgoal     :arg1-reherse-recall-3
   )

(p swap-arg1-reherse-recall-3
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-recall-3
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   =retrieval>
   ISA letters
   slot-number =arg0
   row-number  0
   letter      =arg1text
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   -retrieval>
   =goal>
   subgoal     :arg1-reherse-cleanup-3
   )

(p swap-arg1-reherse-cleanup-3
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal        :arg1-reherse-cleanup-3
   arg0           =arg0
   arg1           =arg1
   arg2           =arg0text
   arg3           =arg1text
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp

   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   - state error
   ==>
   +retrieval>
```

```
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   done           t
   - arg0         :empty
   - arg1         :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1
   problem        =starting-order
   =goal>
   subgoal     :lookup-parent
   dm-reload   :empty
   )

 (p swap-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal      :lookup-parent
   dm-reload    :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg1         :empty
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   done           t
   arg0           :empty
   arg1           :empty
   =goal>
   subgoal        :wait
   )

(p swap-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "swap" 'return-from)
   operator        swap
   subgoal      :wait
   dm-reload    :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "swap" 'dm-recall-parent)
   op-name        swap
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   done           t
   arg0           :empty
   arg1           :empty
```

```
      ?retrieval>
      state free
      - state error
      ==>
      =goal>
      current-branch =return-branch
      branch-order   =return-state
      operator       =return-op
      return-value   :no-value
      arg0           :empty
      arg1           :empty
      arg2           :empy
      arg3           :empty
      subgoal        :empty
      dm-reload      :reload
      next-branch  :empty
      next-branch-number    :empty
      next-operator :empty
      )
  )
)
```

Listing 36: SWAP Operator Listing

B.8 IF-N Operator

```
(define-operator
    :name "if-n"
  :arity 3
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 recall-arg2
                 jump-arg2
                 return-arg2
                 save-arg2
                 done-arg2
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-arg1
                 dm-recall-arg2
                 dm-recall-parent
                 )

  :compiler-for (compiler-sequence-for 'if-n "if-n" 3)
  :productions
      `(
        (p if-n
           =goal>
           ISA metaproc
           current-branch =branch
           branch-order   0
           operator      if-n
           ?retrieval>
           state free
           ==>
           +retrieval>
           ISA op-sequence
           branch-name  =branch
           branch-order ,(jump-state-lookup "if-n" 'dm-recall 0)
```

```
   op-name      if-n
   =goal>
   branch-order ,(jump-state-lookup "if-n" 'recall 0)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'if-n "if-n" 3 0 :type-boolean)

(p if-n-arg0-done-true
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 0)
   operator       if-n
   return-value   true
   subgoal        :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall-parent)
   op-name        if-n
   arg0           =arg0
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "if-n" 'dm-recall 1)
   op-name      if-n
   =goal>
   subgoal      :load
   NEXT-BRANCH    =return-branch
   NEXT-BRANCH-NUMBER  =return-state
   NEXT-OPERATOR  =return-op
   )

(p if-n-arg0-done-true-load
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 0)
   operator       if-n
   return-value   true
   subgoal        :load
   NEXT-BRANCH    =main-return-branch
   NEXT-BRANCH-NUMBER  =main-return-state
   NEXT-OPERATOR  =main-return-op
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order    ,(jump-state-lookup "if-n" 'dm-recall 1)
   op-name        if-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +imaginal>
```

```
   ISA op-sequence
   branch-name    =branch
   branch-order    ,(jump-state-lookup "if-n" 'dm-recall-parent )
   done           t
   op-name        if-n
   arg0           true
   arg1           :empty
   arg2           :empty
   arg3           :empty
   arg4           :empty
   arg5           :empty
   arg6           :empty
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   timestamp      =timestamp
   last-argument  0
   loop-iteration =loop-iteration
   problem       =starting-order
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall 1)
   op-name        if-n
   =goal>
   subgoal :jump
   NEXT-BRANCH    :empty
   NEXT-BRANCH-NUMBER  :empty
   NEXT-OPERATOR   :empty
   )
(p if-n-arg0-done-true-jump
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 0)
   operator       if-n
   return-value   true
   subgoal        :jump
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall 1)
   op-name        if-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   -imaginal>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "if-n" 'dm-recall 1)
   op-name      if-n
   =goal>
   subgoal :empty
   branch-order ,(jump-state-lookup "if-n" 'recall 1)
   )

(p if-n-arg0-done-false
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 0)
   operator       if-n
   return-value   false
   subgoal        :empty
   =retrieval>
```

```
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall-parent)
   op-name        if-n
   arg0           =arg0
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "if-n" 'dm-recall 2 )
   op-name      if-n
   =goal>
   subgoal :load
   NEXT-BRANCH    =return-branch
   NEXT-BRANCH-NUMBER  =return-state
   NEXT-OPERATOR  =return-op
   )
(p if-n-arg0-done-false-load
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 0)
   operator       if-n
   return-value   false
   subgoal        :load
   NEXT-BRANCH    =main-return-branch
   NEXT-BRANCH-NUMBER  =main-return-state
   NEXT-OPERATOR  =main-return-op
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order    ,(jump-state-lookup "if-n" 'dm-recall 2)
   op-name        if-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +imaginal>
   ISA op-sequence
   branch-name    =branch
   branch-order    ,(jump-state-lookup "if-n" 'dm-recall-parent )
   done           t
   op-name        if-n
   arg0           false
   arg1           :no-value
   arg2           :empty
   arg3           :empty
   arg4           :empty
   arg5           :empty
   arg6           :empty
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   timestamp      =timestamp
   last-argument  0
   loop-iteration =loop-iteration
```

```
   problem        =starting-order
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall 2)
   op-name        if-n
   =goal>
   subgoal :jump
   NEXT-BRANCH    :empty
   NEXT-BRANCH-NUMBER  :empty
   NEXT-OPERATOR   :empty
   )
(p if-n-arg0-done-false-jump
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 0)
   operator       if-n
   return-value   false
   subgoal        :jump
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall 2)
   op-name        if-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   -imaginal>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "if-n" 'dm-recall 2)
   op-name      if-n
   =goal>
   subgoal :empty
   branch-order ,(jump-state-lookup "if-n" 'recall 2)
   )

(p if-n-arg0-done-other
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 0)
   operator       if-n
   - return-value true
   - return-value false
   - return-value :empty
   return-value   =return-value
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall-parent)
   op-name        if-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
```

```
   branch-order ,(jump-state-lookup "if-n" 'dm-recall-parent)
   op-name      if-n
   done         t
   =goal>
   branch-order ,(jump-state-lookup "if-n" 'return-from)
   return-value  =return-value
   )

,@(argument-p-sequence-for 'if-n "if-n" 3 1 :type-number nil t)

(p if-n-arg1-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 1)
   operator       if-n
   return-value   =return-value
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall-parent)
   op-name        if-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "if-n" 'dm-recall-parent)
   op-name      if-n
   done         t
   arg0         true
   - arg1       :empty
   arg2         :no-value
   =goal>
   branch-order ,(jump-state-lookup "if-n" 'return-from)
   return-value =return-value
   )

,@(argument-p-sequence-for 'if-n "if-n" 3 2 :type-number nil t)

(p if-n-arg2-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "if-n" 'done 2)
   operator       if-n
   return-value   =return-value
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "if-n" 'dm-recall-parent)
   op-name        if-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "if-n" 'dm-recall-parent)
```

```
            op-name      if-n
            done         t
            arg0         false
            arg1         :no-value
            - arg2       :empty
            =goal>
            current-branch     =branch
            branch-order       ,(jump-state-lookup "if-n" 'return-from)
            return-value       =return-value
            )

         (p if-n-return
            =goal>
            ISA metaproc
            current-branch =branch
            branch-order   ,(jump-state-lookup "if-n" 'return-from)
            operator        if-n
            return-value   =returnval
            starting-order =starting-order
            loop-iteration =loop-iteration
            =retrieval>
            ISA op-sequence
            branch-name    =branch
            branch-order   ,(jump-state-lookup "if-n" 'dm-recall-parent)
            op-name        if-n
            return-branch  =return-branch
            return-state   =return-state
            return-operator =return-op
            ?retrieval>
            state free
            - state error
            ==>
            =goal>
            current-branch =return-branch
            branch-order   =return-state
            operator       =return-op
            return-value   =returnval
            arg0           :empty
            arg1           :empty
            dm-reload      :reload
            next-branch  :empty
            next-branch-number    :empty
            next-operator :empty
            subgoal   :empty
            )
               )
    )
```

Listing 37: IF-N Operator Listing

B.9 ONCE-PER-PROBLEM-N Operator

```
(define-operator
    :name "once-per-problem-n"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-past
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from)
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    dm-recall-past
                    )
    :compiler-for (compiler-sequence-for 'once-per-problem-n "once-per-problem-n" 1)
    :productions
```

```
`(
 (p once-per-problem-n
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   0
    operator       once-per-problem-n
    current-problem =problem
    timestamp       =time
    starting-order  =starting
    ?retrieval>
    state free
    ==>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(jump-state-lookup "once-per-problem-n" 'dm-recall-past)
    op-name      once-per-problem-n
    - arg0       :empty
    - arg1       :empty
    arg1         =time
    - arg2        :empty
    arg2         =starting
    =goal>
    branch-order ,(jump-state-lookup "once-per-problem-n" 'recall-past)
    subgoal      :check
    )

 (p once-per-problem-n-recall-past-success
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(jump-state-lookup "once-per-problem-n" 'recall-past)
    operator       once-per-problem-n
    timestamp      =time
    starting-order  =starting
    current-problem =problem
    return-value         =oldreturn
    subgoal        :check
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(jump-state-lookup "once-per-problem-n" 'dm-recall-past)
    op-name      once-per-problem-n
    arg0         =value
    arg1         =time
    arg2         =starting
    return-branch    =return-branch
    return-state     =return-state
    return-operator  =return-op
    ==>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(jump-state-lookup "once-per-problem-n" 'dm-recall-parent)
    op-name      once-per-problem-n
    done         t
    arg0         :empty
    arg1         :empty
    arg2         :empty
    =goal>
    return-value       =value
    subgoal            :wait
    )

 (p once-per-problem-n-recall-past-success-lookup-parent
    =goal>
    ISA metaproc
    current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "once-per-problem-n" 'recall-past)
   operator       once-per-problem-n
   timestamp      =time
   starting-order  =starting
   current-problem =problem
   return-value    =value
   subgoal        :wait
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "once-per-problem-n" 'dm-recall-parent)
   op-name      once-per-problem-n
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   =goal>
   current-branch     =return-branch
   branch-order       =return-state
   operator           =return-op
   return-value       =value
   arg0               :empty
   dm-reload          :reload
   subgoal            :empty
   )

(p once-per-problem-n-recall-past-failure
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-per-problem-n" 'recall-past)
   operator       once-per-problem-n
   timestamp      =time
   return-value   =oldreturn
   subgoal        :check
   ?retrieval>
   state free
   state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "once-per-problem-n" 'dm-recall 0)
   op-name      once-per-problem-n
   =goal>
   branch-order ,(jump-state-lookup "once-per-problem-n" 'recall 0)
   subgoal      :empty
   )


,@(argument-p-sequence-for 'once-per-problem-n "once-per-problem-n" 1 0 :type-number)

(p once-per-problem-n-arg0-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-per-problem-n" 'done 0)
   operator       once-per-problem-n
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   ?retrieval>
   state free
   - state error
   =retrieval>
```

```
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-per-problem-n" 'dm-recall-parent)
   op-name          once-per-problem-n
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-per-problem-n" 'dm-recall-parent)
   op-name      once-per-problem-n
   - arg0       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  0
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "once-per-problem-n" 'return-from)
   operator           once-per-problem-n
   subgoal   :encode
   dm-reload :reload
   )

(p once-per-problem-n-return-encode
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "once-per-problem-n" 'return-from)
   operator       once-per-problem-n
   subgoal        :encode
   current-problem =problem
   starting-order  =starting
   timestamp       =time
   ?imaginal>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-per-problem-n" 'dm-recall-parent)
   op-name      once-per-problem-n
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-per-problem-n" 'dm-recall-parent)
   op-name          once-per-problem-n
   done             t
   arg0             :empty
   arg1             :empty
   arg2             :empty
   +imaginal>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "once-per-problem-n" 'dm-recall-past)
   op-name          once-per-problem-n
   arg0             =arg0
   arg1             =time
   arg2             =starting
   return-branch    =return-branch
```

```
        return-state     =return-state
        return-operator  =return-op
        =goal>
        subgoal :commit
        )

      (p once-per-problem-n-return-commit
        =goal>
        ISA metaproc
        current-branch =branch
        branch-order   ,(jump-state-lookup "once-per-problem-n" 'return-from)
        operator       once-per-problem-n
        subgoal        :commit
        current-problem =problem
        starting-order  =starting
        timestamp       =time
        return-value    =return-value
        ?imaginal>
        state free
        =imaginal>
        ISA op-sequence
        branch-name   =branch
        branch-order  ,(jump-state-lookup "once-per-problem-n" 'dm-recall-past)
        op-name       once-per-problem-n
        arg0          =arg0
        arg1          =time
        arg2          =starting
        ?retrieval>
        state free
        - state error
        =retrieval>
        ISA op-sequence
        branch-name  =branch
        branch-order  ,(jump-state-lookup "once-per-problem-n" 'dm-recall-parent)
        op-name      once-per-problem-n
        done             t
        arg0             :empty
        arg1             :empty
        arg2             :empty
        return-branch    =return-branch
        return-state     =return-state
        return-operator  =return-op
        ==>
        -imaginal>
        =goal>
        current-branch  =return-branch
        branch-order    =return-state
        operator        =return-op
        subgoal         :empty
        dm-reload       :reload
        )
    )
  )
```

Listing 38: ONCE-PER-PROBLEM-N Operator Listing

B.10 NEXT-NUMBER Operator

```
(define-operator
    :name "next-number"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from
                     do-operator
```

```
                really-return
                )
:dm-jumps     '(dm-recall-arg0
                dm-recall-parent
                )
:compiler-for (compiler-sequence-for 'next-number "next-number" 1)
:productions
`(
  (p next-number
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   0
     operator       next-number
     ?retrieval>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "next-number" 'dm-recall 0)
     op-name      next-number
     =goal>
     branch-order ,(jump-state-lookup "next-number" 'recall 0)
     subgoal      :empty
     )

  ,@(argument-p-sequence-for 'next-number "next-number" 1 0 :type-number)

  (p next-number-arg0-done
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "next-number" 'done 0)
     operator       next-number
     starting-order =starting-order
     loop-iteration =loop-iteration
     timestamp      =timestamp
     ?retrieval>
     state free
     - state error
     =retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
     op-name      next-number
     return-branch    =return-branch
     return-state     =return-state
     return-operator  =return-op
     ?imaginal>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
     op-name      next-number
     - arg0       :empty
     timestamp      =timestamp
     loop-iteration =loop-iteration
     last-argument  0
     problem        =starting-order
     =goal>
     current-branch     =branch
     branch-order       ,(jump-state-lookup "next-number" 'return-from)
     operator           next-number
          dm-reload :reload
     )

  (p next-number-return
     =goal>
     ISA metaproc
```

```
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'return-from)
   operator       next-number
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'do-operator)
   operator           next-number
   return-value       =arg0
   subgoal            :empty
   )

(p next-number-find-start-fail
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   return-value   :no-value
   subgoal        :empty
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       =arg0
   subgoal            :load-parent
   )

(p next-number-note-match-0
   =goal>
   ISA metaproc
```

```
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   0
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       :no-value
   )

(p next-number-note-match-1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   1
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       2
   )

(p next-number-note-match-2
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   2
   subgoal        :empty
```

```
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       3
   )

(p next-number-note-match-3
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   3
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       4
   )

(p next-number-note-match-4
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   4
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
```

```
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name       next-number
   - arg0        :empty
   arg0          =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       5
   )

(p next-number-note-match-5
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   5
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name       next-number
   - arg0        :empty
   arg0          =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       6
   )

(p next-number-note-match-6
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   6
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name       next-number
   - arg0        :empty
   arg0          =arg0
   return-branch    =return-branch
   return-state     =return-state
```

```
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       7
   )

(p next-number-note-match-7
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   7
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       8
   )

(p next-number-note-match-8
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   8
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
```

```
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       9
   )

(p next-number-note-match-9
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   return-value   9
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       0
   )

(p next-number-note-match-too-high
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   > return-value   9
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       :no-value
   )
```

```
(p next-number-note-match-too-low
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'do-operator)
   operator       next-number
   - return-value :no-value
   < return-value   0
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-number" 'really-return)
   operator           next-number
   return-value       :no-value
   )

(p next-number-really-return-load-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-number" 'really-return)
   operator       next-number
   - return-value :empty
   return-value   =return-value
   ?imaginal>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
   op-name      next-number
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   subgoal   :load-parent
   )

(p next-number-really-return
   =goal>
   ISA metaproc
```

```
        current-branch =branch
        branch-order   ,(jump-state-lookup "next-number" 'really-return)
        operator       next-number
        - return-value :empty
        return-value   =return-value
        subgoal        :load-parent
        ?imaginal>
        state free
        =retrieval>
        ISA op-sequence
        branch-name  =branch
        branch-order  ,(jump-state-lookup "next-number" 'dm-recall-parent)
        op-name      next-number
        done         t
        arg0         :empty
        arg1         :empty
        arg2         :empty
        return-branch    =return-branch
        return-state     =return-state
        return-operator  =return-op
        ?retrieval>
        state free
        - state error
        ==>
        =goal>
        current-branch     =return-branch
        branch-order       =return-state
        operator           =return-op
        dm-reload :reload
        subgoal   :empty
        )

      )
    )
```

Listing 39: NEXT-NUMBER Operator Listing

B.11 NEXT-LETTER Operator

```
(define-operator
    :name "next-letter"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from
                     find-top
                     really-return
                     )
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    )
    :compiler-for (compiler-sequence-for 'next-letter "next-letter" 1)
    :productions
    `(
      (p next-letter
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator       next-letter
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
```

```
   branch-name  =branch
   branch-order ,(jump-state-lookup "next-letter" 'dm-recall 0)
   op-name      next-letter
   =goal>
   branch-order ,(jump-state-lookup "next-letter" 'recall 0)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'next-letter "next-letter" 1 0 :type-letter)

(p next-letter-arg0-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter" 'done 0)
   operator       next-letter
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
   op-name      next-letter
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
   op-name      next-letter
   - arg0       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  0
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter" 'return-from)
   operator           next-letter
   dm-reload :reload
   )

(p next-letter-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter" 'return-from)
   operator       next-letter
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
   op-name      next-letter
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
```

```
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter" 'find-top)
   operator           next-letter
   return-value       =arg0
   subgoal            :empty
   )

(p next-letter-find-start-fail
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter" 'find-top)
   operator       next-letter
   return-value   :no-value
   subgoal        :empty
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
   op-name      next-letter
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
   op-name      next-letter
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter" 'really-return)
   operator           next-letter
   return-value       =arg0
   )

(p next-letter-find-start
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter" 'find-top)
   operator       next-letter
   - return-value :no-value
   return-value   =matchthis
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
   op-name      next-letter
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
```

```
   +retrieval>
   ISA alpha-order
   ordinal 1
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter" 'find-top)
   operator           next-letter
   subgoal            :find-pointer
   )

(p next-letter-find-pointer-hit
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter" 'find-top)
   operator       next-letter
   - return-value :no-value
   return-value   =matchthis
   subgoal        :find-pointer
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-order
   letter =matchthis
   ==>
   =retrieval>
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter" 'find-top)
   operator           next-letter
   subgoal            :load-pointer
   )

(p next-letter-find-pointer-miss
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter" 'find-top)
   operator       next-letter
   - return-value :no-value
   return-value   =matchthis
   subgoal        :find-pointer
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-order
   - letter =matchthis
   next =next
   ==>
   +retrieval>
   ISA alpha-order
   letter =next
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter" 'find-top)
   operator           next-letter
   subgoal            :find-pointer
   )


(p next-letter-load-pointer
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter" 'find-top)
   operator       next-letter
   - return-value :no-value
   return-value   =matchthis
   subgoal        :load-pointer
   ?retrieval>
```

```
      state free
      - state error
      =retrieval>
      ISA alpha-order
      letter =return-letter
      next =return-next
      ==>
      +retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
      op-name      next-letter
      done         t
      arg0         :empty
      arg1         :empty
      arg2         :empty
        =goal>
      current-branch     =branch
      branch-order       ,(jump-state-lookup "next-letter" 'really-return)
      operator           next-letter
      subgoal            :empty
      return-value       =return-next
      arg0               :empty
      )

    (p next-letter-really-return
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order   ,(jump-state-lookup "next-letter" 'really-return)
      operator       next-letter
      - return-value :empty
      return-value   =return-value
      ?imaginal>
      state free
      ?retrieval>
      state free
      - state error
      =retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order  ,(jump-state-lookup "next-letter" 'dm-recall-parent)
      op-name      next-letter
      done         t
      arg0         :empty
      arg1         :empty
      arg2         :empty
      return-branch    =return-branch
      return-state     =return-state
      return-operator  =return-op
      ==>
      =goal>
      current-branch     =return-branch
      branch-order       =return-state
      operator           =return-op
      dm-reload :reload
      subgoal   :empty
      )

    )
  )
```

Listing 40: NEXT-LETTER Operator Listing

B.12 SCAN-FOR-CHAR-LR Operator

```
(define-operator
    :name "scan-for-char-lr"
    :arity 1
```

```
:prod-jumps    '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 lookat
                 return-from)
:dm-jumps     '(dm-recall-arg0
                dm-recall-parent
                )
:compiler-for (compiler-sequence-for 'scan-for-char-lr "scan-for-char-lr" 1)
:productions
`(
  (p scan-for-char-lr
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   0
     operator       scan-for-char-lr
     ?retrieval>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "scan-for-char-lr" 'dm-recall 0)
     op-name      scan-for-char-lr
     =goal>
     branch-order ,(jump-state-lookup "scan-for-char-lr" 'recall 0)
     subgoal      :empty
     )

  ,@(argument-p-sequence-for 'scan-for-char-lr "scan-for-char-lr" 1 0 :type-letter)

  (p scan-for-char-lr-arg0-done
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "scan-for-char-lr" 'done 0)
     operator       scan-for-char-lr
     length         =len
     return-value   =matchthis
     starting-order =starting-order
     loop-iteration =loop-iteration
     timestamp      =timestamp
     ?retrieval>
     state free
     - state error
     =retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order  ,(jump-state-lookup "scan-for-char-lr" 'dm-recall-parent)
     op-name      scan-for-char-lr
     return-branch    =return-branch
     return-state     =return-state
     return-operator  =return-op
     ?imaginal>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order  ,(jump-state-lookup "scan-for-char-lr" 'dm-recall-parent)
     op-name      scan-for-char-lr
     - arg0       :empty
     timestamp      =timestamp
     loop-iteration =loop-iteration
     last-argument  0
     problem        =starting-order
     =goal>
     current-branch     =branch
```

```
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            1
   dm-reload          :empty
   arg0               1
   arg1               0
   arg2               =matchthis
   ;dm-reload :reload
   )

,@(screen-pos-literal-generator
   #'(lambda (len x y x-index y-index)
       (let* ((x-lower (- x +X-LOWER-TOLERANCE+))
              (x-upper (+ x +X-UPPER-TOLERANCE+))
              (y-lower (- y +Y-LOWER-TOLERANCE+))
              (y-upper (+ y +Y-UPPER-TOLERANCE+)))
         `(p ,(screen-pos-literal-name-generator
               "scan-for-char-lr" len x-index y-index)
             =goal>
             ISA metaproc
             current-branch     =branch
             branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
             operator           scan-for-char-lr
             subgoal            ,(1+ x-index)
             dm-reload          :empty
             length             ,len
             arg0               ,(1+ x-index)
             arg1               ,y-index
             ?visual-location>
             state free
             ==>
             +visual-location>
             ISA visual-location
             >  screen-x ,x-lower
             <= screen-x ,x-upper
             >  screen-y ,y-lower
             <= screen-y ,y-upper
             kind text
             =goal>
             subgoal     ,(if (> (1+ x-index) len) :lookat-done (1+ x-index) )
             arg0        ,(1+ x-index)
             dm-reload   ,(if (> (1+ x-index) len) :empty :attend)
             )
         )
       )
   )

(p scan-for-char-lr-lookat-dolook
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            =x-index
   dm-reload          :attend
   length             =len
   arg0               =arg0
   arg1               =arg1
   arg2               =arg2
   arg3               =arg3
   arg4               =arg4
   =visual-location>
   ISA visual-location
   screen-x =screenx
   screen-y =screeny
   kind text
   ?visual-location>
   state free
   ?visual>
   state free
   ==>
   +visual>
```

```
   ISA move-attention
   screen-pos =visual-location
   =goal>
   dm-reload :encode
   )

(p scan-for-char-lr-lookat-encode
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            =x-index
   dm-reload          :encode
   length             =len
   arg0               =x
   arg1               =y
   arg2               =matchthis
   arg3               =old-arg3
   arg4               =old-arg4
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?imaginal>
   state free
   ==>
   +imaginal>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   =visual>
   =goal>
   arg3      =textvalue
   arg4      =x
   dm-reload :cleanup
   )

(p scan-for-char-lr-lookat-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            =x-index
   dm-reload          :cleanup
   length             =len
   arg0               =arg0
   arg1               =arg1
   ?imaginal>
   state free
   ==>
   -imaginal>
   =goal>
   dm-reload        :check-match
   )

 (p scan-for-char-lr-lookat-check-matched
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            =x-index
   dm-reload          :check-match
   length             =len
   arg0               =arg0
   arg1               =arg1
   arg2               =arg2
```

```
   arg3              =arg2
   arg4              =storedval
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "scan-for-char-lr" 'dm-recall-parent)
   op-name      scan-for-char-lr
   =goal>
   branch-order     ,(jump-state-lookup "scan-for-char-lr" 'return-from)
   subgoal          :empty
   dm-reload        :empty
   arg0             :empty
   arg1             :empty
   arg2             :empty
   arg3             :empty
   return-value     =storedval
   )

(p scan-for-char-lr-lookat-check-failed-middle
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            =x-index
   dm-reload          :check-match
   length             =len
   arg0               =arg0
   - arg0             =len
   arg1               =arg1
   arg2               =arg2
   arg3               =arg3
   - arg3             =arg2
   arg4               =storedval
   ?imaginal>
   state free
   ==>
   !safe-bind!        =next-step (1+ =x-index)
   -imaginal>
   =goal>
   dm-reload        :empty
   arg0             =next-step
   arg3             :empty
   arg4             :empty
   subgoal          =next-step
   return-value     :no-value
   )

(p scan-for-char-lr-lookat-check-failed-end
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            =x-index
   dm-reload          :check-match
   length             =len
   arg0               =arg0
   arg0               =len
   arg1               =arg1
   arg2               =arg2
   arg3               =arg3
   - arg3             =arg2
   arg4               =storedval
   ?imaginal>
   state free
```

```
   ==>
   -imaginal>
   =goal>
   branch-order     ,(jump-state-lookup "scan-for-char-lr" 'return-from)
   subgoal          :empty
   dm-reload        :empty
   arg0             :empty
   arg1             :empty
   arg2             :empty
   arg3             :empty
   return-value     :no-value
   )

(p scan-for-char-lr-lookat-done
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'lookat)
   operator           scan-for-char-lr
   subgoal            :lookat-done
   dm-reload          :empty
   length             =len
   arg0               =arg0
   arg1               =arg1
   ?imaginal>
   state free
   ==>
   -imaginal>
   =goal>
   branch-order       ,(jump-state-lookup "scan-for-char-lr" 'return-from)
   subgoal            :empty
   dm-reload          :empty
   )

(p scan-for-char-lr-return-load
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "scan-for-char-lr" 'return-from)
   operator       scan-for-char-lr
   subgoal        :empty
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "scan-for-char-lr" 'dm-recall-parent)
   op-name        scan-for-char-lr
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   subgoal      :load
   )

(p scan-for-char-lr-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "scan-for-char-lr" 'return-from)
   operator       scan-for-char-lr
   subgoal        :load
   return-value   =matchthis
   ?imaginal>
   state free
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "scan-for-char-lr" 'dm-recall-parent)
```

```
        op-name      scan-for-char-lr
        done         t
        arg0         :empty
        arg1         :empty
        arg2         :empty
        return-branch    =return-branch
        return-state     =return-state
        return-operator  =return-op
        ?retrieval>
        state free
        - state error
        ==>
        =goal>
        current-branch     =return-branch
        branch-order       =return-state
        operator           =return-op
        dm-reload :reload
        subgoal   :empty
        )

    )
  )
```

Listing 41: SCAN-FOR-CHAR-LR Operator Listing

B.13 SCAN-FOR-NUM-LR Operator

```
(define-operator
    :name "scan-for-num-lr"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     lookat
                     return-from)
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    )
    :compiler-for (compiler-sequence-for 'scan-for-num-lr "scan-for-num-lr" 1)
    :productions
    `(
      (p scan-for-num-lr
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator       scan-for-num-lr
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "scan-for-num-lr" 'dm-recall 0)
         op-name      scan-for-num-lr
         =goal>
         branch-order ,(jump-state-lookup "scan-for-num-lr" 'recall 0)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'scan-for-num-lr "scan-for-num-lr" 1 0 :type-number)

      (p scan-for-num-lr-arg0-done
         =goal>
         ISA metaproc
         current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "scan-for-num-lr" 'done 0)
   operator       scan-for-num-lr
   length         =len
   return-value   =matchthis
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "scan-for-num-lr" 'dm-recall-parent)
   op-name      scan-for-num-lr
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "scan-for-num-lr" 'dm-recall-parent)
   op-name      scan-for-num-lr
   - arg0       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  0
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            1
   dm-reload          :empty
   arg0               1
   arg1               0
   arg2               =matchthis
   )

,@(screen-pos-literal-generator
   #'(lambda (len x y x-index y-index)
       (let* ((x-lower (- x +X-LOWER-TOLERANCE+))
              (x-upper (+ x +X-UPPER-TOLERANCE+))
              (y-lower (- y +Y-LOWER-TOLERANCE+))
              (y-upper (+ y +Y-UPPER-TOLERANCE+)))
         `(p ,(screen-pos-literal-name-generator
               "scan-for-num-lr" len x-index y-index)
             =goal>
             ISA metaproc
             current-branch     =branch
             branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
             operator           scan-for-num-lr
             subgoal            ,(1+ x-index)
             dm-reload          :empty
             length             ,len
             arg0               ,(1+ x-index)
             arg1               ,y-index
             ?visual-location>
             state free
             ==>
             +visual-location>
             ISA visual-location
             >  screen-x ,x-lower
             <= screen-x ,x-upper
             >  screen-y ,y-lower
             <= screen-y ,y-upper
             kind text
             =goal>
             subgoal     ,(if (> (1+ x-index) len) :lookat-done (1+ x-index) )
```

```
            arg0        ,(1+ x-index)
            dm-reload   ,(if (> (1+ x-index) len) :empty :attend)
            )
         )
      )
   )

(p scan-for-num-lr-lookat-dolook
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            =x-index
   dm-reload          :attend
   length             =len
   arg0               =arg0
   arg1               =arg1
   arg2               =arg2
   arg3               =arg3
   arg4               =arg4
   =visual-location>
   ISA visual-location
   screen-x =screenx
   screen-y =screeny
   kind text
   ?visual-location>
   state free
   ?visual>
   state free
   ==>
   +visual>
   ISA move-attention
   screen-pos =visual-location
   =goal>
   dm-reload :encode
   )

(p scan-for-num-lr-lookat-encode
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            =x-index
   dm-reload          :encode
   length             =len
   arg0               =x
   arg1               =y
   arg2               =matchthis
   arg3               =old-arg3
   arg4               =old-arg4
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?imaginal>
   state free
   ==>
   +imaginal>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   =visual>
   =goal>
   arg3      =x
   arg4      =textvalue
   dm-reload :cleanup
   )
```

```
(p scan-for-num-lr-lookat-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            =x-index
   dm-reload          :cleanup
   length             =len
   arg0               =arg0
   arg1               =arg1
   ?imaginal>
   state free
   ==>
   -imaginal>
   =goal>
   dm-reload        :check-match
   )

 (p scan-for-num-lr-lookat-check-matched
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            =x-index
   dm-reload          :check-match
   length             =len
   arg0               =arg0
   arg1               =arg1
   arg2               =arg2
   arg3               =arg2
   arg4               =storedval
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "scan-for-num-lr" 'dm-recall-parent)
   op-name      scan-for-num-lr
   =goal>
   branch-order     ,(jump-state-lookup "scan-for-num-lr" 'return-from)
   subgoal          :empty
   dm-reload        :empty
   arg0             :empty
   arg1             :empty
   arg2             :empty
   arg3             :empty
   return-value     =storedval
   )

(p scan-for-num-lr-lookat-check-failed-middle
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            =x-index
   dm-reload          :check-match
   length             =len
   arg0               =arg0
   - arg0             =len
   arg1               =arg1
   arg2               =arg2
   arg3               =arg3
   - arg3             =arg2
   arg4               =storedval
```

```
   ?imaginal>
   state free
   ==>
   !safe-bind!        =next-step (1+ =x-index)
   -imaginal>
   =goal>
   dm-reload        :empty
   arg0             =next-step
   arg3             :empty
   arg4             :empty
   subgoal          =next-step
   return-value     :no-value
   )

(p scan-for-num-lr-lookat-check-failed-end
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            =x-index
   dm-reload          :check-match
   length             =len
   arg0               =arg0
   arg0               =len
   arg1               =arg1
   arg2               =arg2
   arg3               =arg3
   - arg3             =arg2
   arg4               =storedval
   ?imaginal>
   state free
   ==>
   -imaginal>
   =goal>
   branch-order     ,(jump-state-lookup "scan-for-num-lr" 'return-from)
   subgoal          :empty
   dm-reload        :empty
   arg0             :empty
   arg1             :empty
   arg2             :empty
   arg3             :empty
   return-value     :no-value
   )

(p scan-for-num-lr-lookat-done
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'lookat)
   operator           scan-for-num-lr
   subgoal            :lookat-done
   dm-reload          :empty
   length             =len
   arg0               =arg0
   arg1               =arg1
   ?imaginal>
   state free
   ==>
   -imaginal>
   =goal>
   branch-order       ,(jump-state-lookup "scan-for-num-lr" 'return-from)
   subgoal            :empty
   dm-reload          :empty
   )

(p scan-for-num-lr-return-load
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "scan-for-num-lr" 'return-from)
   operator       scan-for-num-lr
```

```
        subgoal        :empty
        ?retrieval>
        state free
        - state error
        ==>
        +retrieval>
        ISA op-sequence
        branch-name    =branch
        branch-order   ,(jump-state-lookup "scan-for-num-lr" 'dm-recall-parent)
        op-name        scan-for-num-lr
        done          t
        arg0          :empty
        arg1          :empty
        arg2          :empty
        =goal>
        subgoal        :load
        )

      (p scan-for-num-lr-return
        =goal>
        ISA metaproc
        current-branch =branch
        branch-order   ,(jump-state-lookup "scan-for-num-lr" 'return-from)
        operator       scan-for-num-lr
        subgoal        :load
        return-value   =matchthis
        ?imaginal>
        state free
        =retrieval>
        ISA op-sequence
        branch-name  =branch
        branch-order  ,(jump-state-lookup "scan-for-num-lr" 'dm-recall-parent)
        op-name      scan-for-num-lr
        done          t
        arg0          :empty
        arg1          :empty
        arg2          :empty
        return-branch    =return-branch
        return-state     =return-state
        return-operator  =return-op
        ?retrieval>
        state free
        - state error
        ==>
        =goal>
        current-branch     =return-branch
        branch-order       =return-state
        operator           =return-op
        dm-reload :reload
        subgoal   :empty
        )

      )
    )
```

Listing 42: SCAN-FOR-NUM-LR Operator Listing

B.14 LOOK-AT-CHAR Operator

```
(define-operator
    :name "look-at-char"
    :arity 1
    :compiler-for (compiler-sequence-for 'look-at-char "look-at-char" 1)
    :prod-jumps  '(entry-point
                   recall-arg0
                   jump-arg0
                   return-arg0
                   save-arg0
                   done-arg0
```

```
                lookup-coords
                return-from)
:dm-jumps    '(dm-recall-arg0
                dm-recall-parent
                )
:productions
`(
  (p look-at-char
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   0
     operator      look-at-char
     ?retrieval>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "look-at-char" 'dm-recall 0)
     op-name      look-at-char
     =goal>
     branch-order ,(jump-state-lookup "look-at-char" 'recall 0)
     subgoal      :empty
     )

  ,@(argument-p-sequence-for 'look-at-char "look-at-char" 1 0 :type-letter)

  (p look-at-char-arg0-done
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "look-at-char" 'done 0)
     operator       look-at-char
     =retrieval>
     ISA op-sequence
     branch-name    =branch
     branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
     op-name        look-at-char
     return-branch  =return-branch
     return-state   =reutrn-state
     return-operator =return-op
     ?retrieval>
     state free
     - state error
     ?imaginal>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
     op-name      look-at-char
     =goal>
     branch-order ,(jump-state-lookup "look-at-char" 'lookup-coords)
     subgoal      :prepare
   )

  (p look-at-char-prep-coords
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
     operator       look-at-char
     subgoal       :prepare
     dm-reload     :empty
     - return-value :empty
     - return-value :no-value
     return-value  =x
     =retrieval>
     ISA op-sequence
     branch-name    =branch
```

```
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :grab-letter
   dm-reload          :empty
   return-value       :empty
   arg0               =x
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )


(p look-at-char-prep-coords-bad-no0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator       look-at-char
   subgoal       :prepare
   dm-reload     :empty
   return-value  :no-value
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'return-from)
   operator           look-at-char
   subgoal            :empty
   dm-reload          :empty
   return-value       :no-value
   )

(p look-at-char-prep-coords-bad-no0-too-low
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator       look-at-char
```

```
   subgoal       :prepare
   dm-reload     :empty
   - arg0  :no-value
   < arg0  0
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'return-from)
   operator           look-at-char
   subgoal            :empty
   dm-reload          :empty
   return-value       :no-value
   )

(p look-at-char-prep-coords-bad-no0-too-high
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator       look-at-char
   subgoal       :prepare
   dm-reload     :empty
   length        =length
   - arg0  :no-value
   > arg0  =length
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'return-from)
   operator           look-at-char
```

```
   subgoal            :empty ;:grab-letter
   dm-reload          :empty
   return-value       :no-value
   )

(p look-at-char-prep-coords-grab-letter
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator       look-at-char
   subgoal       :grab-letter
   dm-reload     :empty
   arg0          =arg0
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   letter =arg0
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :grab-letter-check
   dm-reload          :empty
   arg0               =arg0
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p look-at-char-prep-coords-grab-letter-check
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator       look-at-char
   subgoal       :grab-letter-check
   dm-reload     :empty
   arg0          =arg0
   =retrieval>
   ISA letters
   letter =arg0
   slot-number =slot
   row-number  =row
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :grab-letter-check-valid
   dm-reload          :empty
   arg0               =arg0
   )

(p look-at-char-prep-coords-grab-letter-check-fail
```

```
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
  operator       look-at-char
  subgoal       :grab-letter-check
  dm-reload     :empty
  arg0          =arg0
  ?retrieval>
  state free
  state error
  ==>
  +retrieval>
  ISA op-sequence
  branch-name    =branch
  branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
  op-name        look-at-char
  done           t
  arg0           :empty
  arg1           :empty
  arg2           :empty
  =goal>
  current-branch     =branch
  branch-order       ,(jump-state-lookup "look-at-char" 'return-from)
  operator           look-at-char
  subgoal            :empty
  dm-reload          :empty
  arg0               :empty
  return-value       :no-value
  )

 (p look-at-char-prep-coords-grab-letter-check-valid
  =goal>
  ISA metaproc
  current-branch =branch
  branch-order   ,(jump-state-lookup "look-at-char" 'lookup-coords)
  operator       look-at-char
  subgoal       :grab-letter-check-valid
  dm-reload     :empty
  arg0          =arg0
  length        =len
  =retrieval>
  ISA letters
  letter =arg0
  >= slot-number 0
  <= slot-number =len
  slot-number =slot
  row-number  =row
  ?retrieval>
  state free
  - state error
  ==>
  =retrieval>
  =goal>
  current-branch     =branch
  branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
  operator           look-at-char
  subgoal            :match
  dm-reload          :empty
  arg0               =slot
  arg1               =row
  )

,@(screen-pos-literal-generator
  #'(lambda (len x y x-index y-index)
      (let* ((x-lower (- x +X-LOWER-TOLERANCE+))
             (x-upper (+ x +X-UPPER-TOLERANCE+))
             (y-lower (- y +Y-LOWER-TOLERANCE+))
             (y-upper (+ y +Y-UPPER-TOLERANCE+)))

        `(p ,(screen-pos-literal-name-generator
              "look-at-char" len x-index y-index)
```

```
            =goal>
            ISA metaproc
            current-branch     =branch
            branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
            operator           look-at-char
            subgoal            :match
            dm-reload          :empty
            length             ,len
            arg0               ,x-index
            arg1               ,y-index
            =retrieval>
            ISA letters
            letter =arg0
            slot-number =slot
            row-number  =row
            ?visual-location>
            state free
            ==>
            =retrieval>
            +visual-location>
            ISA visual-location
            >  screen-x ,x-lower
            <= screen-x ,x-upper
            >  screen-y ,y-lower
            <= screen-y ,y-upper
            kind text
            =goal>
            subgoal     :lookat
            )
        )
      )
   )

(p look-at-char-bad-visual-loc
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :lookat
   dm-reload          :empty
   =retrieval>
   ISA letters
   letter =arg0
   slot-number =slot
   row-number  =row
   ?visual-location>
   state free
   state error
   ==>
   =retrieval>
   =goal>
   subgoal     :cleanup
   dm-reload   :empty
   return-value :no-value
   )

(p look-at-char-lookat
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :lookat
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   =visual-location>
   ISA visual-location
   screen-x =screenx
   screen-y =screeny
```

```
   kind text
   ?visual-location>
   state free
   ?visual>
   state free
   ==>
   +visual>
   ISA move-attention
   screen-pos =visual-location
   =goal>
   subgoal :encode
   )

(p look-at-char-encode
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :encode
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?imaginal>
   state free
   ==>
   +imaginal>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   =visual>
   =goal>
   subgoal :cleanup
   return-value =x
   )

(p look-at-char-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :cleanup
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   -imaginal>
   =visual>
   =goal>
```

```
   branch-order ,(jump-state-lookup "look-at-char" 'lookup-coords)
   subgoal :rehersal
   )

(p look-at-char-rehersal
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :rehersal
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   =visual>
   ISA text
   value =textvalue
   ?visual>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   +retrieval>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   =visual>
   =goal>
   subgoal :rehersal-cleanup
   )

(p look-at-char-rehersal-cleanup
   =goal>
   ISA metaproc
   current-branch     =branch
   branch-order       ,(jump-state-lookup "look-at-char" 'lookup-coords)
   operator           look-at-char
   subgoal            :rehersal-cleanup
   dm-reload          :empty
   length             =len
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   =retrieval>
   ISA letters
   letter =textvalue
   slot-number =x
   row-number  =y
   done        :empty
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   -imaginal>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
   op-name        look-at-char
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   done         t
   arg0         :empty
```

```
        arg1         :empty
        arg2         :empty
        =goal>
        branch-order ,(jump-state-lookup "look-at-char" 'return-from)
        subgoal :empty
        arg0    :empty
        arg1    :empty
        )

      (p look-at-char-return
        =goal>
        ISA metaproc
        current-branch =branch
        branch-order   ,(jump-state-lookup "look-at-char" 'return-from)
        operator        look-at-char
        return-value    =returnval
        =retrieval>
        ISA op-sequence
        branch-name    =branch
        branch-order   ,(jump-state-lookup "look-at-char" 'dm-recall-parent)
        op-name        look-at-char
        done          t
        arg0          :empty
        arg1          :empty
        arg2          :empty
        return-branch  =return-branch
        return-state   =return-state
        return-operator =return-op
        ?retrieval>
        state free
        - state error
        ==>
        =goal>
        current-branch =return-branch
        branch-order   =return-state
        operator       =return-op
        arg0           :empty
        arg1           :empty
        dm-reload      :reload
        next-branch  :empty
        next-branch-number    :empty
        next-operator :empty
        subgoal   :empty
        )

      )

    )
```

Listing 43: LOOK-AT-CHAR Operator Listing

B.15 NEXT-LETTER-IN-SONG Operator

```
(define-operator
    :name "next-letter-in-song"
    :arity 1
    :prod-jumps    '(entry-point
                     recall-arg0
                     jump-arg0
                     return-arg0
                     save-arg0
                     done-arg0
                     return-from
                     find-top
                     really-return
                     )
    :dm-jumps     '(dm-recall-arg0
                    dm-recall-parent
                    )
```

```
:compiler-for (compiler-sequence-for 'next-letter-in-song "next-letter-in-song" 1)
:productions
`(
  (p next-letter-in-song
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   0
     operator       next-letter-in-song
     ?retrieval>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order ,(jump-state-lookup "next-letter-in-song" 'dm-recall 0)
     op-name      next-letter-in-song
     =goal>
     branch-order ,(jump-state-lookup "next-letter-in-song" 'recall 0)
     subgoal      :empty
     )

  ,@(argument-p-sequence-for 'next-letter-in-song "next-letter-in-song" 1 0 :type-letter)

  (p next-letter-in-song-arg0-done
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "next-letter-in-song" 'done 0)
     operator       next-letter-in-song
     starting-order =starting-order
     loop-iteration =loop-iteration
     timestamp      =timestamp
     ?retrieval>
     state free
     - state error
     =retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
     op-name      next-letter-in-song
     return-branch    =return-branch
     return-state     =return-state
     return-operator  =return-op
     ?imaginal>
     state free
     ==>
     +retrieval>
     ISA op-sequence
     branch-name  =branch
     branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
     op-name      next-letter-in-song
     - arg0       :empty
     timestamp      =timestamp
     loop-iteration =loop-iteration
     last-argument  0
     problem        =starting-order
     =goal>
     current-branch     =branch
     branch-order       ,(jump-state-lookup "next-letter-in-song" 'return-from)
     operator           next-letter-in-song
     dm-reload :reload
     )

  (p next-letter-in-song-return
     =goal>
     ISA metaproc
     current-branch =branch
     branch-order   ,(jump-state-lookup "next-letter-in-song" 'return-from)
     operator       next-letter-in-song
     ?imaginal>
     state free
```

```
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
   op-name      next-letter-in-song
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator           next-letter-in-song
   return-value       =arg0
   subgoal            :empty
   )

(p next-letter-in-song-find-start-fail
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   return-value   :no-value
   subgoal        :empty
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
   op-name      next-letter-in-song
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
   op-name      next-letter-in-song
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'really-return)
   operator           next-letter-in-song
   return-value       =arg0
   )

(p next-letter-in-song-find-start
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :empty
```

```
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
   op-name      next-letter-in-song
   - arg0       :empty
   arg0         =arg0
   return-branch    =return-branch
   return-state     =return-state
   return-operator  =return-op
   ==>
   +retrieval>
   ISA alpha-order
   letter =matchthis
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator           next-letter-in-song
   subgoal            :lookup-ordinal
   )

(p next-letter-in-song-find-lookup-ord
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :lookup-ordinal
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-order
   letter =matchthis
   ordinal =ordinal
   ==>
   +retrieval>
   ISA alpha-song-chunk
   <= min =ordinal
   >= max =ordinal
   type :top
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator           next-letter-in-song
   subgoal            :find-top
   arg0               =ordinal
   )

(p next-letter-in-song-find-find-first-pointer
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :find-top
   - arg0         :empty
   arg0           =arg0
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-song-chunk
   <= min =arg0
   >= max =arg0
```

```
   type :top
   - key-letter :empty
   key-letter =key-letter
   ==>
   +retrieval>
   ISA alpha-song-chunk
   key-letter =key-letter
   type :pointer
   target =key-letter
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator           next-letter-in-song
   subgoal            :find-pointer
   )

(p next-letter-in-song-find-pointer-hit
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :find-pointer
   - arg0         :empty
   arg0           =arg0
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-song-chunk
   <= min =arg0
   >= max =arg0
   type :pointer
   - target :empty
   target =matchthis
   - next-song-chunk-key :no-value
   ==>
   +retrieval>
   ISA alpha-order
   prev =matchthis
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator           next-letter-in-song
   subgoal            :load-pointer
   )

(p next-letter-in-song-find-pointer-hit-out
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :find-pointer
   - arg0         :empty
   arg0           =arg0
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-song-chunk
   <= min =arg0
   >= max =arg0
   type :pointer
   - target :empty
   target =matchthis
   next-song-chunk-key :no-value
   ==>
```

```
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
   op-name      next-letter-in-song
   done t
   arg0 :empty
   arg1 :empty
   arg2 :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'really-return)
   operator           next-letter-in-song
   return-value       :no-value
   subgoal            :empty
   arg0               :empty
   )

(p next-letter-in-song-find-pointer-miss
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :find-pointer
   - arg0         :empty
   arg0           =arg0
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-song-chunk
   <= min =arg0
   >= max =arg0
   type :pointer
   - target :empty
   - target =matchthis
   - next-song-chunk-key :no-value
   next-song-chunk-key =nextletter
   ==>
   +retrieval>
   ISA alpha-song-chunk
   target =nextletter
   type  :pointer
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator           next-letter-in-song
   subgoal            :find-pointer
   )

(p next-letter-in-song-find-pointer-miss-out
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :find-pointer
   - arg0         :empty
   arg0           =arg0
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-song-chunk
   <= min =arg0
   >= max =arg0
   type :pointer
```

```
   - target :empty
   - target =matchthis
   next-song-chunk-key :no-value
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
   op-name      next-letter-in-song
   - arg0       :empty
     =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'really-return)
   operator           next-letter-in-song
   return-value       :no-value
   subgoal            :empty
   arg0               :empty
   )


(p next-letter-in-song-load-pointer
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'find-top)
   operator       next-letter-in-song
   - return-value :no-value
   return-value   =matchthis
   subgoal        :load-pointer
   - arg0         :empty
   arg0           =arg0
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA alpha-order
   letter =return-letter
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
   op-name      next-letter-in-song
   done         t
   arg0         :empty
   arg1         :empty
   arg2         :empty
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "next-letter-in-song" 'really-return)
   operator           next-letter-in-song
   subgoal            :empty
   return-value       =return-letter
   arg0               :empty
   )

(p next-letter-in-song-really-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "next-letter-in-song" 'really-return)
   operator       next-letter-in-song
   - return-value :empty
   return-value   =return-value
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   =retrieval>
   ISA op-sequence
   branch-name  =branch
```

```
        branch-order  ,(jump-state-lookup "next-letter-in-song" 'dm-recall-parent)
        op-name      next-letter-in-song
        done         t
        arg0         :empty
        arg1         :empty
        arg2         :empty
        return-branch    =return-branch
        return-state     =return-state
        return-operator  =return-op
        ==>
        =goal>
        current-branch     =return-branch
        branch-order       =return-state
        operator           =return-op
        dm-reload :reload
        subgoal   :empty
        )

    )
  )
```

Listing 44: NEXT-LETTER-IN-SONG Operator Listing

B.16 AND Operator

```
(define-operator
    :name "and"
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-arg1
                 dm-recall-parent
                 )

  :compiler-for (compiler-sequence-for 'and "and" 2)

  :productions
      `(
        (p and
           =goal>
           ISA metaproc
           current-branch =branch
           branch-order   0
           operator      and
           ?retrieval>
           state free
           ==>
           +retrieval>
           ISA op-sequence
           branch-name  =branch
           branch-order ,(jump-state-lookup "and" 'dm-recall 0)
           op-name      and
           =goal>
           branch-order ,(jump-state-lookup "and" 'recall 0)
           subgoal      :empty
           )
```

```
,@(argument-p-sequence-for 'and "and" 2 0 :type-boolean)

(p and-arg0-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'done 0)
   operator       and
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "and" 'dm-recall 1)
   op-name      and
   =goal>
   branch-order ,(jump-state-lookup "and" 'recall 1)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'and "and" 2 1 :type-boolean)

(p and-arg1-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'done 1)
   operator       and
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name      and
   done         t
   - arg0       :empty
   - arg1       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1 ;;ARG0,1 both done
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "and" 'do-operation)
```

```
   operator           and
   subgoal            :prepare
   dm-reload          :empty
   )

(p and-prep-args
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'do-operation)
   operator       and
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "and" 'do-operation)
   operator           and
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p and-match-t-t
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'do-operation)
   operator       and
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           true
   arg1           true
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
```

```
   current-branch     =branch
   branch-order       ,(jump-state-lookup "and" 'return-from)
   operator           and
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p and-match-false-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'do-operation)
   operator       and
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           false
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "and" 'return-from)
   operator           and
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p and-match-false-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'do-operation)
   operator       and
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
```

```
   arg1           false
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "and" 'return-from)
   operator           and
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p and-match-bad-bools-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'do-operation)
   operator       and
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         false
   - arg0         true
   arg0           =x
   - arg1         :empty
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "and" 'return-from)
   operator           and
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p and-match-bad-bools-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'do-operation)
   operator       and
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
```

```
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           =x
   - arg1         :empty
   - arg1         false
   - arg1         true
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "and" 'return-from)
   operator           and
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p and-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "and" 'return-from)
   operator        and
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   - arg1         :empty
   arg1 =arg1
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
   op-name        and
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   subgoal   :wait
   )

(p and-return
   =goal>
   ISA metaproc
   current-branch =branch
```

```
          branch-order   ,(jump-state-lookup "and" 'return-from)
          operator        and
          return-value   =returnval
          subgoal        :wait
          =retrieval>
          ISA op-sequence
          branch-name    =branch
          branch-order   ,(jump-state-lookup "and" 'dm-recall-parent)
          op-name        and
          done           t
          arg0           :empty
          arg1           :empty
          arg2           :empty
          return-branch  =return-branch
          return-state   =return-state
          return-operator =return-op
          ?retrieval>
          state free
          - state error
          ==>
          =goal>
          current-branch =return-branch
          branch-order   =return-state
          operator       =return-op
          return-value   =returnval
          arg0           :empty
          arg1           :empty
          dm-reload      :reload
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )

      )

  )
```

Listing 45: AND Operator Listing

B.17 OR Operator

```
(define-operator
    :name "or"
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-arg1
                 dm-recall-parent
                 )

  :compiler-for (compiler-sequence-for 'or "or" 2)

  :productions
      `(
        (p or
           =goal>
```

```
   ISA metaproc
   current-branch =branch
   branch-order   0
   operator      or
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "or" 'dm-recall 0)
   op-name      or
   =goal>
   branch-order ,(jump-state-lookup "or" 'recall 0)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'or "or" 2 0 :type-boolean)

(p or-arg0-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'done 0)
   operator       or
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "or" 'dm-recall 1)
   op-name      or
   =goal>
   branch-order ,(jump-state-lookup "or" 'recall 1)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'or "or" 2 1 :type-boolean)

(p or-arg1-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'done 1)
   operator       or
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
```

```
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name      or
   done         t
   - arg0       :empty
   - arg1       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1 ;;ARGO,1 both done
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "or" 'do-operation)
   operator           or
   subgoal            :prepare
   dm-reload          :empty
   )

(p or-prep-args
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'do-operation)
   operator       or
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "or" 'do-operation)
   operator           or
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p or-match-f-f
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'do-operation)
   operator       or
   subgoal       :match
   dm-reload     :empty
```

```
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           false
   arg1           false
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "or" 'return-from)
   operator           or
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p or-match-t-t
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'do-operation)
   operator       or
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           true
   arg1           true
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "or" 'return-from)
   operator           or
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p or-match-true-arg0
   =goal>
   ISA metaproc
   current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "or" 'do-operation)
   operator       or
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           true
   arg1           false
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "or" 'return-from)
   operator           or
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p or-match-true-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'do-operation)
   operator       or
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           false
   arg1           true
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "or" 'return-from)
   operator           or
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )
```

```
(p or-match-bad-bools-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'do-operation)
   operator       or
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         false
   - arg0         true
   arg0           =x
   - arg1         :empty
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "or" 'return-from)
   operator           or
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p or-match-bad-bools-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'do-operation)
   operator       or
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           =x
   - arg1         :empty
   - arg1         false
   - arg1         true
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
```

```
   branch-order       ,(jump-state-lookup "or" 'return-from)
   operator           or
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p or-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'return-from)
   operator        or
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   - arg1         :empty
   arg1 =arg1
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   subgoal   :wait
   )

(p or-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "or" 'return-from)
   operator        or
   return-value   =returnval
   subgoal        :wait
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "or" 'dm-recall-parent)
   op-name        or
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ==>
   =goal>
```

```
          current-branch =return-branch
          branch-order   =return-state
          operator       =return-op
          return-value   =returnval
          arg0           :empty
          arg1           :empty
          dm-reload      :reload
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )

       )

     )
```

Listing 46: OR Operator Listing

B.18 NOT Operator

```
(define-operator
    :name "not"
  :arity 1
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-parent
                 )

  :compiler-for (compiler-sequence-for 'not "not" 1)

  :productions
      `(
        (p not
           =goal>
           ISA metaproc
           current-branch =branch
           branch-order   0
           operator      not
           ?retrieval>
           state free
           ==>
           +retrieval>
           ISA op-sequence
           branch-name  =branch
           branch-order ,(jump-state-lookup "not" 'dm-recall 0)
           op-name      not
           =goal>
           branch-order ,(jump-state-lookup "not" 'recall 0)
           subgoal      :empty
           )

        ,@(argument-p-sequence-for 'not "not" 1 0 :type-boolean)

        (p not-arg0-done
           =goal>
           ISA metaproc
           current-branch =branch
           branch-order   ,(jump-state-lookup "not" 'done 0)
           operator       not
           starting-order =starting-order
           loop-iteration =loop-iteration
```

```
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name        not
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name      not
   - arg0       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  0
   problem        =starting-order
   =goal>
   branch-order ,(jump-state-lookup "not" 'do-operation)
   subgoal :prepare
   )

(p not-prep-args
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "not" 'do-operation)
   operator       not
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name        not
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "not" 'do-operation)
   operator           not
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p not-match-f
   =goal>
   ISA metaproc
   current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "not" 'do-operation)
   operator       not
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name        not
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           false
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "not" 'return-from)
   operator           not
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p not-match-t
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "not" 'do-operation)
   operator       not
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name        not
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           true
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "not" 'return-from)
   operator           not
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p not-match-bad-bools
   =goal>
   ISA metaproc
   current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "not" 'do-operation)
   operator       not
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name        not
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         false
   - arg0         true
   arg0           =x
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "not" 'return-from)
   operator           not
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p not-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "not" 'return-from)
   operator        not
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name        not
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
   op-name        not
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   subgoal   :wait
   )

(p not-return
   =goal>
   ISA metaproc
   current-branch =branch
```

```
          branch-order   ,(jump-state-lookup "not" 'return-from)
          operator        not
          return-value   =returnval
          subgoal        :wait
          =retrieval>
          ISA op-sequence
          branch-name    =branch
          branch-order   ,(jump-state-lookup "not" 'dm-recall-parent)
          op-name        not
          done           t
          arg0           :empty
          arg1           :empty
          arg2           :empty
          return-branch  =return-branch
          return-state   =return-state
          return-operator =return-op
          ?retrieval>
          state free
          - state error
          ==>
          =goal>
          current-branch =return-branch
          branch-order   =return-state
          operator       =return-op
          return-value   =returnval
          arg0           :empty
          arg1           :empty
          dm-reload      :reload
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )


       )

     )
```

Listing 47: NOT Operator Listing

B.19 XOR Operator

```
(define-operator
    :name "xor"
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-arg1
                 dm-recall-parent
                 )

  :compiler-for (compiler-sequence-for 'xor "xor" 2)
  :productions
      `(
        (p xor
           =goal>
```

```
   ISA metaproc
   current-branch =branch
   branch-order   0
   operator      xor
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "xor" 'dm-recall 0)
   op-name      xor
   =goal>
   branch-order ,(jump-state-lookup "xor" 'recall 0)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'xor "xor" 2 0 :type-boolean)

(p xor-arg0-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'done 0)
   operator       xor
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "xor" 'dm-recall 1)
   op-name      xor
   =goal>
   branch-order ,(jump-state-lookup "xor" 'recall 1)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'xor "xor" 2 1 :type-boolean)

(p xor-arg1-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'done 1)
   operator       xor
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
```

```
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name      xor
   done         t
   - arg0       :empty
   - arg1       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1 ;;ARGO,1 both done
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "xor" 'do-operation)
   operator           xor
   subgoal            :prepare
   dm-reload          :empty
   )

 (p xor-prep-args
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'do-operation)
   operator       xor
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "xor" 'do-operation)
   operator           xor
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p xor-match-f-f
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'do-operation)
   operator       xor
   subgoal       :match
   dm-reload     :empty
```

```
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           false
   arg1           false
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "xor" 'return-from)
   operator           xor
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p xor-match-t-t
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'do-operation)
   operator       xor
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           true
   arg1           true
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "xor" 'return-from)
   operator           xor
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p xor-match-true-arg0
   =goal>
   ISA metaproc
   current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "xor" 'do-operation)
   operator       xor
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           true
   arg1           false
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "xor" 'return-from)
   operator           xor
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p xor-match-true-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'do-operation)
   operator       xor
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           false
   arg1           true
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "xor" 'return-from)
   operator           xor
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )
```

```
(p xor-match-bad-bools-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'do-operation)
   operator       xor
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         false
   - arg0         true
   arg0           =x
   - arg1         :empty
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "xor" 'return-from)
   operator           xor
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p xor-match-bad-bools-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'do-operation)
   operator       xor
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           =x
   - arg1         :empty
   - arg1         false
   - arg1         true
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
```

```
   branch-order       ,(jump-state-lookup "xor" 'return-from)
   operator           xor
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p xor-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'return-from)
   operator        xor
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   - arg1         :empty
   arg1 =arg1
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   subgoal   :wait
   )

(p xor-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "xor" 'return-from)
   operator        xor
   return-value   =returnval
   subgoal        :wait
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "xor" 'dm-recall-parent)
   op-name        xor
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ==>
   =goal>
```

```
          current-branch =return-branch
          branch-order   =return-state
          operator       =return-op
          return-value   =returnval
          arg0           :empty
          arg1           :empty
          dm-reload      :reload
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )

       )

     )
```

Listing 48: XOR Operator Listing

B.20 NUM< Operator

```
(define-operator
    :name "num<"
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-arg1
                 dm-recall-parent
                 )
  :compiler-for (compiler-sequence-for 'num< "num<" 2)
  :productions
      `(
       (p num<
          =goal>
          ISA metaproc
          current-branch =branch
          branch-order   0
          operator      num<
          ?retrieval>
          state free
          ==>
          +retrieval>
          ISA op-sequence
          branch-name  =branch
          branch-order ,(jump-state-lookup "num<" 'dm-recall 0)
          op-name      num<
          =goal>
          branch-order ,(jump-state-lookup "num<" 'recall 0)
          subgoal      :empty
          )

       ,@(argument-p-sequence-for 'num< "num<" 2 0 :type-number)

       (p num<-arg0-done
          =goal>
          ISA metaproc
          current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "num<" 'done 0)
   operator       num<
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "num<" 'dm-recall 1)
   op-name      num<
   =goal>
   branch-order ,(jump-state-lookup "num<" 'recall 1)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'num< "num<" 2 1 :type-number)

(p num<-arg1-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'done 1)
   operator       num<
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name      num<
   done         t
   - arg0       :empty
   - arg1       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1 ;;ARGO,1 both done
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num<" 'do-operation)
   operator           num<
   subgoal            :prepare
   dm-reload          :empty
   )

(p num<-prep-args
```

```
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'do-operation)
   operator       num<
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num<" 'do-operation)
   operator           num<
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num<-match-arg0-less
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'do-operation)
   operator       num<
   arg0           =arg0
   arg1           =arg1
   < arg0         =arg1
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num<" 'return-from)
   operator           num<
   return-value       true
```

```
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num<-match-arg1-less
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'do-operation)
   operator       num<
   arg0           =arg0
   arg1           =arg1
   < arg1         =arg0
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num<" 'return-from)
   operator           num<
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num<-match-args-eq
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'do-operation)
   operator       num<
   arg0           =arg0
   arg1           =arg0
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
```

```
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num<" 'return-from)
   operator           num<
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num<-match-bad-char-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'do-operation)
   operator       num<
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           :no-value
   arg0           =x
   - arg1         :empty
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num<" 'return-from)
   operator           num<
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num<-match-bad-char-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'do-operation)
   operator       num<
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =return-state
```

```
   return-operator =return-op
   - arg0          :empty
   arg0            =x
   arg1            :no-value
   arg1            =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num<" 'return-from)
   operator           num<
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )


(p num<-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'return-from)
   operator        num<
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   - arg1         :empty
   arg1 =arg1
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
   op-name        num<
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   subgoal   :wait
   )

(p num<-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num<" 'return-from)
   operator        num<
   return-value   =returnval
   subgoal        :wait
   =retrieval>
   ISA op-sequence
```

```
          branch-name    =branch
          branch-order   ,(jump-state-lookup "num<" 'dm-recall-parent)
          op-name        num<
          done           t
          arg0           :empty
          arg1           :empty
          arg2           :empty
          return-branch  =return-branch
          return-state   =return-state
          return-operator =return-op
          ?retrieval>
          state free
          - state error
          ==>
          =goal>
          current-branch =return-branch
          branch-order   =return-state
          operator       =return-op
          return-value   =returnval
          arg0           :empty
          arg1           :empty
          dm-reload      :reload
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )
        )
      )
```

Listing 49: NUM< Operator Listing

B.21 NUM> Operator

```
(define-operator
    :name "num>"
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-arg1
                 dm-recall-parent
                 )
  :compiler-for (compiler-sequence-for 'num> "num>" 2)
  :productions
      `(
        (p num>
           =goal>
           ISA metaproc
           current-branch =branch
           branch-order   0
           operator      num>
           ?retrieval>
           state free
           ==>
           +retrieval>
           ISA op-sequence
           branch-name  =branch
```

```
   branch-order ,(jump-state-lookup "num>" 'dm-recall 0)
   op-name      num>
   =goal>
   branch-order ,(jump-state-lookup "num>" 'recall 0)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'num> "num>" 2 0 :type-number)

(p num>-arg0-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'done 0)
   operator       num>
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "num>" 'dm-recall 1)
   op-name      num>
   =goal>
   branch-order ,(jump-state-lookup "num>" 'recall 1)
   subgoal      :empty
   )

,@(argument-p-sequence-for 'num> "num>" 2 1 :type-number)

(p num>-arg1-done
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'done 1)
   operator       num>
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name      num>
   done         t
   - arg0       :empty
   - arg1       :empty
```

```
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1 ;;ARG0,1 both done
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num>" 'do-operation)
   operator           num>
   subgoal            :prepare
   dm-reload          :empty
   )

(p num>-prep-args
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'do-operation)
   operator       num>
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num>" 'do-operation)
   operator           num>
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num>-match-arg0-more
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'do-operation)
   operator       num>
   arg0           =arg0
   arg1           =arg1
   > arg0         =arg1
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =return-state
```

```
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num>" 'return-from)
   operator           num>
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num>-match-arg1-more
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'do-operation)
   operator       num>
   arg0           =arg0
   arg1           =arg1
   > arg1         =arg0
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num>" 'return-from)
   operator           num>
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num>-match-args-eq
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'do-operation)
   operator       num>
   arg0           =arg0
   arg1           =arg0
   subgoal       :match
   dm-reload     :empty
```

```
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num>" 'return-from)
   operator           num>
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num>-match-bad-char-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'do-operation)
   operator       num>
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           :no-value
   arg0           =x
   - arg1         :empty
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num>" 'return-from)
   operator           num>
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num>-match-bad-char-arg1
   =goal>
```

```
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'do-operation)
   operator       num>
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           =x
   arg1           :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num>" 'return-from)
   operator           num>
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num>-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num>" 'return-from)
   operator        num>
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   - arg1         :empty
   arg1 =arg1
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
   op-name        num>
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   subgoal   :wait
```

```
          )

        (p num>-return
           =goal>
           ISA metaproc
           current-branch =branch
           branch-order   ,(jump-state-lookup "num>" 'return-from)
           operator        num>
           return-value   =returnval
           subgoal        :wait
           =retrieval>
           ISA op-sequence
           branch-name    =branch
           branch-order   ,(jump-state-lookup "num>" 'dm-recall-parent)
           op-name        num>
           done           t
           arg0           :empty
           arg1           :empty
           arg2           :empty
           return-branch  =return-branch
           return-state   =return-state
           return-operator =return-op
           ?retrieval>
           state free
           - state error
           ==>
           =goal>
           current-branch =return-branch
           branch-order   =return-state
           operator       =return-op
           return-value   =returnval
           arg0           :empty
           arg1           :empty
           dm-reload      :reload
           next-branch  :empty
           next-branch-number    :empty
           next-operator :empty
           subgoal   :empty
           )

       )

     )
```

Listing 50: NUM> Operator Listing

B.22 NUM= Operator

```
(define-operator
    :name "num="
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
                 done-arg0
                 recall-arg1
                 jump-arg1
                 return-arg1
                 save-arg1
                 done-arg1
                 do-operation
                 return-from)
  :dm-jumps    '(dm-recall-arg0
                 dm-recall-arg1
                 dm-recall-parent
                 )
  :compiler-for (compiler-sequence-for 'num= "num=" 2)
```

```
:productions
    `(
      (p num=
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator      num=
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "num=" 'dm-recall 0)
         op-name      num=
         =goal>
         branch-order ,(jump-state-lookup "num=" 'recall 0)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'num= "num=" 2 0 :type-number)

      (p num=-arg0-done
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "num=" 'done 0)
         operator       num=
         =retrieval>
         ISA op-sequence
         branch-name    =branch
         branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
         op-name        num=
         return-branch  =return-branch
         return-state   =reutrn-state
         return-operator =return-op
         ?retrieval>
         state free
         - state error
         ?imaginal>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "num=" 'dm-recall 1)
         op-name      num=
         =goal>
         branch-order ,(jump-state-lookup "num=" 'recall 1)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'num= "num=" 2 1 :type-number)

      (p num=-arg1-done
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "num=" 'done 1)
         operator       num=
         starting-order =starting-order
         loop-iteration =loop-iteration
         timestamp      =timestamp
         =retrieval>
         ISA op-sequence
         branch-name    =branch
         branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
         op-name        num=
         return-branch  =return-branch
         return-state   =reutrn-state
         return-operator =return-op
```

```
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name      num=
   done         t
   - arg0       :empty
   - arg1       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1 ;;ARG0,1 both done
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num=" 'do-operation)
   operator           num=
   subgoal            :prepare
   dm-reload          :empty
   )

(p num=-prep-args;;TODO
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num=" 'do-operation)
   operator       num=
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name        num=
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num=" 'do-operation)
   operator           num=
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num=-match-arg0-less
   =goal>
   ISA metaproc
   current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "num=" 'do-operation)
   operator       num=
   arg0           =arg0
   arg1           =arg1
   < arg0         =arg1
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name        num=
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num=" 'return-from)
   operator           num=
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num=-match-arg1-less
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num=" 'do-operation)
   operator       num=
   arg0           =arg0
   arg1           =arg1
   < arg1         =arg0
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name        num=
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num=" 'return-from)
   operator           num=
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
```

```
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num=-match-args-eq
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num=" 'do-operation)
   operator       num=
   arg0           =arg0
   arg1           =arg0
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name        num=
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num=" 'return-from)
   operator           num=
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num=-match-bad-char-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num=" 'do-operation)
   operator       num=
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name        num=
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   arg0           :no-value
   arg0           =x
   - arg1         :empty
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
```

```
   branch-order       ,(jump-state-lookup "num=" 'return-from)
   operator           num=
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num=-match-bad-char-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num=" 'do-operation)
   operator       num=
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name        num=
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           =x
   arg1           :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "num=" 'return-from)
   operator           num=
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p num=-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "num=" 'return-from)
   operator        num=
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
   op-name        num=
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   - arg1         :empty
   arg1 =arg1
```

```
          ?retrieval>
          state free
          ==>
          +retrieval>
          ISA op-sequence
          branch-name    =branch
          branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
          op-name        num=
          done           t
          arg0           :empty
          arg1           :empty
          arg2           :empty
          =goal>
          subgoal   :wait
          )

       (p num=-return
          =goal>
          ISA metaproc
          current-branch =branch
          branch-order   ,(jump-state-lookup "num=" 'return-from)
          operator        num=
          return-value   =returnval
          subgoal        :wait
          =retrieval>
          ISA op-sequence
          branch-name    =branch
          branch-order   ,(jump-state-lookup "num=" 'dm-recall-parent)
          op-name        num=
          done           t
          arg0           :empty
          arg1           :empty
          arg2           :empty
          return-branch  =return-branch
          return-state   =return-state
          return-operator =return-op
          ?retrieval>
          state free
          - state error
          ==>
          =goal>
          current-branch =return-branch
          branch-order   =return-state
          operator       =return-op
          return-value   =returnval
          arg0           :empty
          arg1           :empty
          dm-reload      :reload
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )
       )
     )
```

Listing 51: NUM= Operator Listing

B.23 ASSERT-N Operator

```
(when (and *debug-mode* *run-testing-harness*)
  (define-operator
    :name "assert-n"
  :arity 2
  :prod-jumps  '(entry-point
                 recall-arg0
                 jump-arg0
                 return-arg0
                 save-arg0
```

```
                done-arg0
                recall-arg1
                jump-arg1
                return-arg1
                save-arg1
                done-arg1
                do-operation
                return-from)
:dm-jumps    '(dm-recall-arg0
                dm-recall-arg1
                dm-recall-parent
                )
:compiler-for (compiler-sequence-for 'assert-n "assert-n" 2)
:productions
    `(
      (p assert-n
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   0
         operator      assert-n
         ?retrieval>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "assert-n" 'dm-recall 0)
         op-name      assert-n
         =goal>
         branch-order ,(jump-state-lookup "assert-n" 'recall 0)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'assert-n "assert-n" 2 0 :type-number)

      (p assert-n-arg0-done
         =goal>
         ISA metaproc
         current-branch =branch
         branch-order   ,(jump-state-lookup "assert-n" 'done 0)
         operator       assert-n
         =retrieval>
         ISA op-sequence
         branch-name    =branch
         branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
         op-name        assert-n
         return-branch  =return-branch
         return-state   =reutrn-state
         return-operator =return-op
         ?retrieval>
         state free
         - state error
         ?imaginal>
         state free
         ==>
         +retrieval>
         ISA op-sequence
         branch-name  =branch
         branch-order ,(jump-state-lookup "assert-n" 'dm-recall 1)
         op-name      assert-n
         =goal>
         branch-order ,(jump-state-lookup "assert-n" 'recall 1)
         subgoal      :empty
         )

      ,@(argument-p-sequence-for 'assert-n "assert-n" 2 1 :type-number)

      (p assert-n-arg1-done
         =goal>
         ISA metaproc
         current-branch =branch
```

```
   branch-order   ,(jump-state-lookup "assert-n" 'done 1)
   operator       assert-n
   starting-order =starting-order
   loop-iteration =loop-iteration
   timestamp      =timestamp
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =reutrn-state
   return-operator =return-op
   ?retrieval>
   state free
   - state error
   ?imaginal>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name  =branch
   branch-order ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name      assert-n
   done         t
   - arg0       :empty
   - arg1       :empty
   timestamp      =timestamp
   loop-iteration =loop-iteration
   last-argument  1
   problem        =starting-order
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "assert-n" 'do-operation)
   operator           assert-n
   subgoal            :prepare
   dm-reload          :empty
   )

(p assert-n-prep-args
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
   operator       assert-n
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   - arg0         :no-value
   arg0           =x
   - arg1         :empty
   - arg1         :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (format t "DEBUG: ASSERT-N[~S] has args: arg0='~S', and arg1='~S'~%" =branch =x =y)
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "assert-n" 'do-operation)
```

```
   operator           assert-n
   subgoal            :match
   dm-reload          :empty
   arg0               =x
   arg1               =y
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p assert-n-match-arg0-less
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
   operator       assert-n
   arg0           =arg0
   arg1           =arg1
   < arg0         =arg1
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (progn (format t "DEBUG: ASSERT-N: arg0<arg1; Halting immediately~%")
           (finish-output)
           (when *assert-halts-immediately*(experiment-halt nil t)))
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "assert-n" 'return-from)
   operator           assert-n
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p assert-n-match-arg1-less
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
   operator       assert-n
   arg0           =arg0
   arg1           =arg1
   < arg1         =arg0
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
```

```
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (progn (format t "DEBUG: ASSERT-N: arg0>arg1; Halting immediately~%")
           (finish-output)
           (when *assert-halts-immediately*(experiment-halt nil t)))
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "assert-n" 'return-from)
   operator           assert-n
   return-value       false
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p assert-n-match-args-eq
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
   operator       assert-n
   arg0           =arg0
   arg1           =arg0
   subgoal       :match
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   !eval!         (numberp =arg0)
   ==>
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "assert-n" 'return-from)
   operator           assert-n
   return-value       true
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p assert-n-match-bad-num-arg0
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
   operator       assert-n
   subgoal       :prepare
   dm-reload     :empty
```

```
    =retrieval>
    ISA op-sequence
    branch-name    =branch
    branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
    op-name        assert-n
    return-branch  =return-branch
    return-state   =return-state
    return-operator =return-op
    - arg0           :no-value
    arg0           =x
    - arg1         :empty
    arg1           =y
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    !eval!         (not (numberp =x))
    ==>
    !eval! (progn (format t "DEBUG: ASSERT-N: arg0[~S] is not a number, has type [~S]; Halting immediately~%" =x
    ↪  (type-of =x))
           (finish-output)
           (when *assert-halts-immediately*(experiment-halt nil t)))
    =retrieval>
    =goal>
    current-branch     =branch
    branch-order       ,(jump-state-lookup "assert-n" 'return-from)
    operator           assert-n
    return-value       :no-value
    subgoal            :empty
    dm-reload          :empty
    arg0               :empty
    arg1               :empty
    next-branch  =return-branch
    next-branch-number   =return-state
    next-operator =return-op
    )

(p assert-n-match-bad-num-noval-arg0
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
    operator       assert-n
    subgoal       :prepare
    dm-reload     :empty
    =retrieval>
    ISA op-sequence
    branch-name    =branch
    branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
    op-name        assert-n
    return-branch  =return-branch
    return-state   =return-state
    return-operator =return-op
    arg0           :no-value
    arg0           =x
    - arg1         :empty
    arg1           =y
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    ==>
    !eval! (progn (format t "DEBUG: ASSERT-N: arg0=:no-value; Halting immediately~%")
           (finish-output)
           (when *assert-halts-immediately*(experiment-halt nil t)))
    =retrieval>
    =goal>
    current-branch     =branch
    branch-order       ,(jump-state-lookup "assert-n" 'return-from)
    operator           assert-n
```

```
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p assert-n-match-bad-num-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
   operator       assert-n
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0           =x
   - arg1           :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   !eval!        (not (numberp =y))
   ==>
   !eval! (progn (format t "DEBUG: ASSERT-N: arg1[~S] is not a number, has type [~S]; Halting immediately~%" =y
   ↪  (type-of =y))
         (finish-output)
         (when *assert-halts-immediately*(experiment-halt nil t)))
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "assert-n" 'return-from)
   operator           assert-n
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p assert-n-match-bad-num-noval-arg1
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'do-operation)
   operator       assert-n
   subgoal       :prepare
   dm-reload     :empty
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
```

```
   - arg0         :empty
   arg0           =x
   arg1           :no-value
   arg1           =y
   ?imaginal>
   state free
   ?retrieval>
   state free
   - state error
   ==>
   !eval! (progn (format t "DEBUG: ASSERT-N: arg1=:no-value; Halting immediately~%")
          (finish-output)
          (when *assert-halts-immediately*(experiment-halt nil t)))
   =retrieval>
   =goal>
   current-branch     =branch
   branch-order       ,(jump-state-lookup "assert-n" 'return-from)
   operator           assert-n
   return-value       :no-value
   subgoal            :empty
   dm-reload          :empty
   arg0               :empty
   arg1               :empty
   next-branch  =return-branch
   next-branch-number   =return-state
   next-operator =return-op
   )

(p assert-n-return-lookup-parent
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'return-from)
   operator        assert-n
   return-value   =returnval
   =retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   return-branch  =return-branch
   return-state   =return-state
   return-operator =return-op
   - arg0         :empty
   arg0 =arg0
   - arg1         :empty
   arg1 =arg1
   ?retrieval>
   state free
   ==>
   +retrieval>
   ISA op-sequence
   branch-name    =branch
   branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
   op-name        assert-n
   done           t
   arg0           :empty
   arg1           :empty
   arg2           :empty
   =goal>
   subgoal   :wait
   )

(p assert-n-return
   =goal>
   ISA metaproc
   current-branch =branch
   branch-order   ,(jump-state-lookup "assert-n" 'return-from)
   operator        assert-n
   return-value   =returnval
   subgoal        :wait
   =retrieval>
```

```
          ISA op-sequence
          branch-name    =branch
          branch-order   ,(jump-state-lookup "assert-n" 'dm-recall-parent)
          op-name        assert-n
          done           t
          arg0           :empty
          arg1           :empty
          arg2           :empty
          return-branch  =return-branch
          return-state   =return-state
          return-operator =return-op
          ?retrieval>
          state free
          - state error
          ==>
          ,@(log-return-value "assert-n" '=returnval)
          !eval! (format t "DEBUG: assert-n:~a~%" =returnval)
          =goal>
          current-branch =return-branch
          branch-order   =return-state
          operator       =return-op
          return-value   =returnval
          arg0           :empty
          arg1           :empty
          dm-reload      :reload
          next-branch  :empty
          next-branch-number    :empty
          next-operator :empty
          subgoal   :empty
          )
       )
     )
   )
```

Listing 52: ASSERT-N Operator Listing

B.24 ACT-R Hard-Coded DM Elements

```
(add-dm
       (letter-a ISA alpha-order letter "a" next "b" prev :no-value ordinal 1)
       (letter-b ISA alpha-order letter "b" next "c" prev "a" ordinal 2)
       (letter-c ISA alpha-order letter "c" next "d" prev "b" ordinal 3)
       (letter-d ISA alpha-order letter "d" next "e" prev "c" ordinal 4)
       (letter-e ISA alpha-order letter "e" next "f" prev "d" ordinal 5)
       (letter-f ISA alpha-order letter "f" next "g" prev "e" ordinal 6)
       (letter-g ISA alpha-order letter "g" next "h" prev "f" ordinal 7)
       (letter-h ISA alpha-order letter "h" next "i" prev "g" ordinal 8)
       (letter-i ISA alpha-order letter "i" next "j" prev "h" ordinal 9)
       (letter-j ISA alpha-order letter "j" next "k" prev "i" ordinal 10)
       (letter-k ISA alpha-order letter "k" next "l" prev "j" ordinal 11)
       (letter-l ISA alpha-order letter "l" next "m" prev "k" ordinal 12)
       (letter-m ISA alpha-order letter "m" next "n" prev "l" ordinal 13)
       (letter-n ISA alpha-order letter "n" next "o" prev "m" ordinal 14)
       (letter-o ISA alpha-order letter "o" next "p" prev "n" ordinal 15)
       (letter-p ISA alpha-order letter "p" next "q" prev "o" ordinal 16)
       (letter-q ISA alpha-order letter "q" next "r" prev "p" ordinal 17)
       (letter-r ISA alpha-order letter "r" next "s" prev "q" ordinal 18)
       (letter-s ISA alpha-order letter "s" next "t" prev "r" ordinal 19)
       (letter-t ISA alpha-order letter "t" next "u" prev "s" ordinal 20)
       (letter-u ISA alpha-order letter "u" next "v" prev "t" ordinal 21)
       (letter-v ISA alpha-order letter "v" next "w" prev "u" ordinal 22)
       (letter-w ISA alpha-order letter "w" next "x" prev "v" ordinal 23)
       (letter-x ISA alpha-order letter "x" next "y" prev "w" ordinal 24)
       (letter-y ISA alpha-order letter "y" next "z" prev "x" ordinal 25)
       (letter-z ISA alpha-order letter "z" next :no-value prev "y" ordinal 26)

       (song-alpha ISA alpha-song-chunk
                   start                       true ;; true/false values
                   end                        false       ;;
```

```
            key-letter               "a" ;; the access-point letter
            song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
            ↪  contents of the chunk
            next-song-chunk-key        "h"
            type                  :top
            min                   1
            max                   7
            target                "a"
            )
(song-beta ISA alpha-song-chunk
           start                false
           end                 false
           key-letter         "h"
           song-chunk-sequence         '("h" "i" "j" "k")
           next-song-chunk-key         "l"
           type                  :top
           min                   8
           max                   11
           target                "h"
           )
(song-gamma ISA alpha-song-chunk
            start                 false
            end                  false
            key-letter         "l"
            song-chunk-sequence         '("l" "m" "n" "o" "p")
            next-song-chunk-key         "q"
            type                  :top
            min                   12
            max                   16
            target                "l"
            )
(song-delta ISA alpha-song-chunk
            start                 false
            end                  false
            key-letter         "q"
            song-chunk-sequence         '("q" "r" "s" "t")
            next-song-chunk-key         "u"
            type                  :top
            min                   17
            max                   20
            target                "q"
            )
(song-epsilon ISA alpha-song-chunk
              start                 false
              end                  false
              key-letter         "u"
              song-chunk-sequence         '("u" "v")
              next-song-chunk-key         "w"
              type                  :top
              min                   21
              max                   22
              target                "u"
              )
(song-zeta ISA alpha-song-chunk
           start                false
           end                 true
           key-letter         "w"
           song-chunk-sequence         '("w" "x" "y" "z")
           next-song-chunk-key         :no-value
           type                  :top
           min                   23
           max                   26
           target                "w"
           )

(song-alpha-a ISA alpha-song-chunk
              start                       true ;; true/false values
              end                        false       ;;
              key-letter                "a" ;; the access-point letter
              song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
              ↪  contents of the chunk
              next-song-chunk-key        "b"
```

```
               type                :pointer
               min                 1
               max                 7
               target              "a"
               )
(song-alpha-b ISA alpha-song-chunk
               start                       true ;; true/false values
               end                        false     ;;
               key-letter                 "a" ;; the access-point letter
               song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
               ↪  contents of the chunk
               next-song-chunk-key        "c"
               type                :pointer
               min                 1
               max                 7
               target              "b"
               )
(song-alpha-c ISA alpha-song-chunk
               start                       true ;; true/false values
               end                        false       ;;
               key-letter                 "a" ;; the access-point letter
               song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
               ↪  contents of the chunk
               next-song-chunk-key        "d"
               type                :pointer
               min                 1
               max                 7
               target              "c"
               )
(song-alpha-d ISA alpha-song-chunk
               start                       true ;; true/false values
               end                        false       ;;
               key-letter                 "a" ;; the access-point letter
               song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
               ↪  contents of the chunk
               next-song-chunk-key        "e"
               type                :pointer
               min                 1
               max                 7
               target              "d"
               )
(song-alpha-e ISA alpha-song-chunk
               start                       true ;; true/false values
               end                        false       ;;
               key-letter                 "a" ;; the access-point letter
               song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
               ↪  contents of the chunk
               next-song-chunk-key        "f"
               type                :pointer
               min                 1
               max                 7
               target              "e"
               )
(song-alpha-f ISA alpha-song-chunk
               start                       true ;; true/false values
               end                        false       ;;
               key-letter                 "a" ;; the access-point letter
               song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
               ↪  contents of the chunk
               next-song-chunk-key        "g"
               type                :pointer
               min                 1
               max                 7
               target              "f"
               )
(song-alpha-g ISA alpha-song-chunk
               start                       true ;; true/false values
               end                        true       ;;
               key-letter                 "a" ;; the access-point letter
               song-chunk-sequence          '("a" "b" "c" "d" "e" "f" "g") ;; a list of strings that are the
               ↪  contents of the chunk
               next-song-chunk-key        :no-value
```

```
               type                 :pointer
               min                  1
               max                  7
               target               "g"
               )
(song-beta-h ISA alpha-song-chunk
             start                false
             end                 false
             key-letter         "h"
             song-chunk-sequence        '("h" "i" "j" "k")
             next-song-chunk-key        "i"
             type                 :pointer
             min                  8
             max                  11
             target               "h"
             )
(song-beta-i ISA alpha-song-chunk
             start                false
             end                 false
             key-letter         "h"
             song-chunk-sequence        '("h" "i" "j" "k")
             next-song-chunk-key        "j"
             type                 :pointer
             min                  8
             max                  11
             target               "i"
             )
(song-beta-j ISA alpha-song-chunk
             start                false
             end                 false
             key-letter         "h"
             song-chunk-sequence        '("h" "i" "j" "k")
             next-song-chunk-key        "k"
             type                 :pointer
             min                  8
             max                  11
             target               "j"
             )
(song-beta-k ISA alpha-song-chunk
             start                false
             end                 true
             key-letter         "h"
             song-chunk-sequence        '("h" "i" "j" "k")
             next-song-chunk-key         :no-value
             type                 :pointer
             min                  8
             max                  11
             target               "k"
             )
(song-gamma-l ISA alpha-song-chunk
              start                false
              end                 false
              key-letter         "l"
              song-chunk-sequence        '("l" "m" "n" "o" "p")
              next-song-chunk-key        "m"
              type                 :pointer
              min                  12
              max                  16
              target               "l"
              )
(song-gamma-m ISA alpha-song-chunk
              start                false
              end                 false
              key-letter         "l"
              song-chunk-sequence        '("l" "m" "n" "o" "p")
              next-song-chunk-key        "n"
              type                 :pointer
              min                  12
              max                  16
              target               "m"
              )
(song-gamma-n ISA alpha-song-chunk
```

```
                start                false
                end                 false
                key-letter         "l"
                song-chunk-sequence         '("l" "m" "n" "o" "p")
                next-song-chunk-key         "o"
                type                :pointer
                min                 12
                max                 16
                target              "n"
                )
        (song-gamma-o ISA alpha-song-chunk
                start                false
                end                 false
                key-letter         "l"
                song-chunk-sequence         '("l" "m" "n" "o" "p")
                next-song-chunk-key         "p"
                type                :pointer
                min                 12
                max                 16
                target              "o"
                )
        (song-gamma-p ISA alpha-song-chunk
                start                false
                end                 true
                key-letter         "l"
                song-chunk-sequence         '("l" "m" "n" "o" "p")
                next-song-chunk-key         :no-value
                type                :pointer
                min                 12
                max                 16
                target              "p"
                )
        (song-delta-q ISA alpha-song-chunk
                start                false
                end                 false
                key-letter         "q"
                song-chunk-sequence         '("q" "r" "s" "t")
                next-song-chunk-key         "r"
                type                :pointer
                min                 17
                max                 20
                target              "q"
                )
         (song-delta-r ISA alpha-song-chunk
                start                false
                end                 false
                key-letter         "q"
                song-chunk-sequence         '("q" "r" "s" "t")
                next-song-chunk-key         "s"
                type                :pointer
                min                 17
                max                 20
                target              "r"
                )
          (song-delta-s ISA alpha-song-chunk
                start                false
                end                 false
                key-letter         "q"
                song-chunk-sequence         '("q" "r" "s" "t")
                next-song-chunk-key         "t"
                type                :pointer
                min                 17
                max                 20
                target              "s"
                )
           (song-delta-t ISA alpha-song-chunk
                start                false
                end                 true
                key-letter         "q"
                song-chunk-sequence         '("q" "r" "s" "t")
                next-song-chunk-key         :no-value
                type                :pointer
```

```
          min                  17
          max                  20
          target               "t"
          )

(song-epsilon-u ISA alpha-song-chunk
          start                false
          end                 false
          key-letter          "u"
          song-chunk-sequence         '("u" "v")
          next-song-chunk-key         "v"
          type                 :pointer
          min                  21
          max                  22
          target               "u"
          )
 (song-epsilon-v ISA alpha-song-chunk
          start                false
          end                 true
          key-letter          "u"
          song-chunk-sequence         '("u" "v")
          next-song-chunk-key         :no-value
          type                 :pointer
          min                  21
          max                  22
          target               "v"
          )

 (song-zeta-w ISA alpha-song-chunk
       start                false
       end                 false
       key-letter          "w"
       song-chunk-sequence         '("w" "x" "y" "z")
       next-song-chunk-key         "x"
       type                 :pointer
       min                  23
       max                  26
       target               "w"
       )

 (song-zeta-x ISA alpha-song-chunk
       start                false
       end                 false
       key-letter          "w"
       song-chunk-sequence         '("w" "x" "y" "z")
       next-song-chunk-key         "y"
       type                 :pointer
       min                  23
       max                  26
       target               "x"
       )
 (song-zeta-y ISA alpha-song-chunk
       start                false
       end                 false
       key-letter          "w"
       song-chunk-sequence         '("w" "x" "y" "z")
       next-song-chunk-key         "z"
       type                 :pointer
       min                  23
       max                  26
       target               "y"
       )
 (song-zeta-z ISA alpha-song-chunk
       start                false
       end                 true
       key-letter          "w"
       song-chunk-sequence         '("w" "x" "y" "z")
       next-song-chunk-key         :no-value
       type                 :pointer
       min                  23
       max                  26
       target               "z"
```

```
                )


        (starting-state ISA  metaproc
                        current-problem    ,*current-problem*
                        starting-order     ,(copy-seq *current-problem*)
                        last-problem       :empty
                        length             ,(length *current-problem*)
                        current-branch     :empty
                        branch-order       :empty
                        subgoal            :start-timer
                        next-branch        :empty
                        next-branch-number :empty
                        next-operator      :empty
                        loop-iteration     0
                        operator     :empty
                        arg0     :empty
                        arg1     :empty
                        arg2     :empty
                        arg3     :empty
                        arg4     :empty
                        arg5     :empty
                        arg6     :empty
                        dm-reload :empty
                        timestamp 0 ;this is default time
                        time-since-process-start 0
                        time-since-last-break    0
                        time-current             0
                        )



        ,@dm-list

        )
```

Listing 53: ACT-R Declarative Memory Elemets

B.25 ARGUMENT-P-SEQUENCE-FOR Code

```
(defun argument-p-sequence-for (operator-sym
                                operator-str
                                arg-count
                                arg-num

                                &optional (literal-type :type-letter)
                                  (control-op nil)
                                  (path-control-operator nil)
                                  )
  (let
      ((debug-result
         (flet ((prod-name-fn (step)
                  (dsl-string-to-symbol (format nil "~a-ARG~d-~a" (string-upcase operator-str) arg-num (string-upcase
                  ↪  step))))
                  (prod-jump-fn (step &optional (ignore-arg-num nil))
                    (let ((result
                           (if ignore-arg-num
                               (jump-state-lookup operator-str step)
                               (jump-state-lookup operator-str step arg-num))))

                      result
                      )
                    )
                  )
           (let* ((arg-slot (dsl-string-to-symbol (format nil "arg~d" arg-num)))
                  (arg-slot-binding (dsl-string-to-symbol (format nil "=~s" arg-slot)))
                  (prod-slots-before-me (loop for i from 0 below arg-num
                                              appending
                                                (let* ((current-arg-slot (dsl-string-to-symbol (format nil "arg~d" i)))
```

```
                                    )
                                  `(- ,current-arg-slot :empty)
                                  )
                                )
        )
       (prod-slots-before-me-binding
        (loop for item in
             (loop for i from 0 below arg-num
                collecting
                  (let* ((current-arg-slot (dsl-string-to-symbol (format nil "arg~d" i)))
                         )
                    `(- ,current-arg-slot :empty)
                    )
                  )

           appending                          ; collecting
             (let* ((item-slot (car (cdr item)))
                    (item-binding-name
                     (dsl-string-to-symbol (format nil "=~a" item-slot)))
                    (binding-part (list item-slot item-binding-name))
                    )
               (append item binding-part))
             )
        )
       (prod-slots-save-match-now (append prod-slots-before-me
                                          `(,arg-slot :empty)
                                          ))
       (prod-slots-save-match-now-fn (lambda (x)
                                       (append prod-slots-before-me `(,arg-slot ,x))
                                       ))

       (prod-slots-after-me (loop for i from (1+ arg-num) below arg-count
                                  appending
                                    (let* ((current-arg-slot (dsl-string-to-symbol (format nil "arg~d" i)))
                                           )
                                      `(,current-arg-slot :empty)
                                      )
                                    ))
       )

(append
 (list
  ;;productions

  `(p ,(prod-name-fn "all-prepare")
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order ,(prod-jump-fn 'recall)
      operator  ,operator-sym
      ?retrieval>
      state free
      - state error
      =retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order ,(prod-jump-fn 'dm-recall)
      op-name   ,operator-sym
      ,arg-slot     ,arg-slot-binding
      return-branch =return-branch
      return-state  =return-state
      return-operator =return-op
      ==>
      =retrieval>
      =goal>
      current-branch     =branch
      branch-order      ,(prod-jump-fn 'jump)
      operator          ,operator-sym
      return-value       :empty
      )

  `(p ,(prod-name-fn "all-recall-subexpr")
```

```
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order ,(prod-jump-fn 'jump)
    operator  ,operator-sym
    return-value   :empty
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall)
    op-name      ,operator-sym
    ,arg-slot     :empty
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-op
    ==>
    ;;allow the dm element to be garbage collected
    ;;because we're going to jump to elsewhere in
    ;;a minute anyway. Don't do this in the literal
    ;;case handler
    =goal>
    current-branch   =return-branch
    branch-order     =return-state
    operator         =return-op
    )

`(p ,(prod-name-fn "all-recall-literal")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order ,(prod-jump-fn 'jump)
    operator     ,operator-sym
    return-value :empty
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall)
    op-name      ,operator-sym
    - ,arg-slot  :empty
    ,arg-slot    =argvalue
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-op
    ==>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@(per-arg-do arg-num :empty :processed)
    =goal>
    subgoal  :save
    return-value =argvalue
    )

`(p ,(prod-name-fn "all-recall-literal-save-literal")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order ,(prod-jump-fn 'jump)
    operator     ,operator-sym
    subgoal      :save
    return-value =return-value
    starting-order =starting-order
    loop-iteration =loop-iteration
    timestamp      =timestamp
```

```
      ?retrieval>
      state free
      - state error
      =retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order ,(prod-jump-fn 'dm-recall-parent t)
      op-name      ,operator-sym
      ,@(per-arg-do arg-num :empty :bind)
      done =done
      return-branch =return-branch
      return-state  =return-state
      return-operator =return-operator
      ?imaginal>
      state free
      ==>
      =retrieval>
      +imaginal>
      ISA op-sequence
      branch-name  =branch
      branch-order ,(prod-jump-fn 'dm-recall-parent t)
      op-name      ,operator-sym
      done         =done
      ,@(per-arg-do arg-num :fill :bind '=return-value nil)
      return-branch =return-branch
      return-state  =return-state
      return-operator =return-operator
      timestamp      ,(if (zerop arg-num) :no-value '=timestamp)
      last-argument  ,(if (zerop arg-num) :no-value (1- arg-num))
      loop-iteration ,(if (zerop arg-num) :no-value '=loop-iteration)
      problem        =starting-order
      =goal>
      subgoal  :commit
      )

  `(p ,(prod-name-fn "all-recall-literal-commit")
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order ,(prod-jump-fn 'jump)
      operator     ,operator-sym
      subgoal      :commit
      ?imaginal>
      state free
      =retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order ,(prod-jump-fn 'dm-recall-parent t)
      op-name      ,operator-sym
      ?retrieval>
      state free ;;safe to ommit - state error, because no modification could have happened
      ==>
      -imaginal>
      +retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order ,(prod-jump-fn 'dm-recall)
      op-name      ,operator-sym
      =goal>
      branch-order ,(prod-jump-fn 'return)
      subgoal   :empty
      dm-reload :reload
      )
  )

 (case literal-type
   (:type-letter
    (list
     `(p ,(prod-name-fn "L-all-subexpr-to-literal")
         =goal>
         ISA metaproc
         current-branch =branch
```

```
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    return-value   =return-value
    dm-reload      :reload
    subgoal        :empty
    loop-iteration =loop-iteration
    starting-order =starting-order
    timestamp      =timestamp
    ?retrieval>
    state free
    - state error
    ==>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@prod-slots-save-match-now
    timestamp      ,(if (zerop arg-num) :no-value '=timestamp)
    last-argument  ,(if (zerop arg-num) :no-value (1- arg-num))
    loop-iteration ,(if (zerop arg-num) :no-value '=loop-iteration)
    =goal>
    return-value      =return-value
    current-branch    =branch
    branch-order      ,(prod-jump-fn 'jump)
    operator          ,operator-sym
    dm-reload         :empty
    )

`(p ,(prod-name-fn "L-literal-from-subexpr")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'jump)
    operator      ,operator-sym
    subgoal :empty
    - return-value  :empty
    - return-value  :no-value
    return-value    =return-value
    subgoal         :empty
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order  ,(prod-jump-fn 'dm-recall-parent t)
    op-name       ,operator-sym
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-op
    ==>
    =goal>
    return-value       =return-value
    branch-order       ,(prod-jump-fn 'return)
    operator           ,operator-sym
    current-branch     =branch
    )
`(p ,(prod-name-fn "L-literal-from-subexpr-fallthrough")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'jump)
    operator      ,operator-sym
    return-value  :no-value
    subgoal       :empty
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
```

```
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,arg-slot        :empty
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-op
    ==>
    =retrieval>
    =goal>
    return-value       :no-value
    branch-order       ,(prod-jump-fn 'return)
    current-branch     =branch
    )

`(p ,(prod-name-fn "L-letter-ok")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    dm-reload          =dm-reload
    return-value       =return-value
    ?retrieval>
    state free
    - state error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@(per-arg-do arg-num :empty :processed)
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :commit
    )

`(p ,(prod-name-fn "L-letter-ok-save")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    return-value   =return-value
    dm-reload      :commit
    starting-order =starting-order
    loop-iteration =loop-iteration
    timestamp =timestamp
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         =done
    ,@(per-arg-do arg-num :empty :bind)
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    ==>
```

```
    +imaginal>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         =done
    ,@(per-arg-do arg-num :fill
                  (if path-control-operator :fill :bind)
                  '=return-value nil "=arg-d" t
                  (when path-control-operator  arg-count)
                  (when path-control-operator '=arg0))
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    timestamp      =timestamp
    loop-iteration =loop-iteration
    last-argument ,(if path-control-operator (1- arg-count) arg-num)
    problem       =starting-order
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :clear
    )

`(p ,(prod-name-fn "L-letter-ok-commit")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value   =return-value
    dm-reload      :clear
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    ==>
    -imaginal>
    =retrieval>
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'done)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :empty
    )

`(p ,(prod-name-fn "L-letter-fail")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value       =old
    - dm-reload    :reload
```

```
    ?retrieval>
    state  free
    state  error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@prod-slots-save-match-now
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =old
    dm-reload          :fail-commit
    )

`(p ,(prod-name-fn "L-letter-fail-novalue")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value       :no-value
    - dm-reload    :reload
    ?retrieval>
    state  free
    - state error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@prod-slots-save-match-now
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       :no-value
    dm-reload          :fail-commit
    )

`(p ,(prod-name-fn "L-letter-fail-save")
        =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =next-return-branch
    next-branch-number =next-return-state
    return-value   =return-value
    dm-reload      :fail-commit
    starting-order =starting-order
    loop-iteration =loop-iteration
    timestamp =timestamp
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
```

```
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         t
    ,@(per-arg-do arg-num :empty :bind)
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    ==>
    +imaginal>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         t;=done
    ,@(per-arg-do arg-num :fill
                  (if path-control-operator :fill :bind)
                  '=return-value nil "=arg~d" t
                  (when path-control-operator  arg-count)
                  (when path-control-operator '=arg0))
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    loop-iteration =loop-iteration
    timestamp      =timestamp
    last-argument ,arg-num
    problem       =starting-order
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :fail-clear
    )

`(p ,(prod-name-fn "L-letter-fail-commit")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value   =return-value
    dm-reload      :fail-clear
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    ==>
    -imaginal>
    =retrieval>
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'done)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :empty
    )

)
```

```
 )
(:type-number
 (list
  `(p ,(prod-name-fn "N-all-subexpr-to-literal")
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order   ,(prod-jump-fn 'return)
      operator       ,operator-sym
      return-value   =return-value
      dm-reload      :reload
      subgoal        :empty
      loop-iteration =loop-iteration
      starting-order =starting-order
      timestamp      =timestamp
      ?retrieval>
      state free
      - state error
      ==>
      +retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order ,(prod-jump-fn 'dm-recall-parent t)
      op-name      ,operator-sym
      ,@prod-slots-save-match-now
      timestamp      ,(if (zerop arg-num) :no-value '=timestamp)
      last-argument  ,(if (zerop arg-num) :no-value (1- arg-num))
      loop-iteration ,(if (zerop arg-num) :no-value '=loop-iteration)
      =goal>
      return-value      =return-value
      current-branch    =branch
      branch-order      ,(prod-jump-fn 'jump)
      operator          ,operator-sym
      dm-reload         :empty
      )

  `(p ,(prod-name-fn "N-literal-from-subexpr")
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order   ,(prod-jump-fn 'jump)
      operator      ,operator-sym
      - return-value  :empty
      - return-value  :no-value
      return-value    =return-value
      subgoal         :empty
      ?retrieval>
      state free
      - state error
      =retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order  ,(prod-jump-fn 'dm-recall-parent t)
      op-name       ,operator-sym ;index-of-letter
      return-branch =return-branch
      return-state  =return-state
      return-operator =return-op
      ==>
      =goal>
      return-value       =return-value ;:empty
      branch-order       ,(prod-jump-fn 'return)
      operator           ,operator-sym
      current-branch     =branch
      )
  `(p ,(prod-name-fn "N-literal-from-subexpr-fallthrough")
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order   ,(prod-jump-fn 'jump)
      operator      ,operator-sym
      return-value  :no-value
      subgoal       :empty
```

```
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym ;index-of-letter
    ,arg-slot        :empty
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-op
    ==>
    =retrieval>
    =goal>
    return-value       :no-value
    branch-order       ,(prod-jump-fn 'return)
    current-branch     =branch
    )

`(p ,(prod-name-fn "N-num-ok")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    dm-reload      =dm-reload
    return-value   =return-value
    ?retrieval>
    state free
    - state error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@(per-arg-do arg-num :empty :processed)
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :commit
    )

`(p ,(prod-name-fn "N-num-ok-save")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    return-value   =return-value
    dm-reload      :commit
    starting-order =starting-order
    loop-iteration =loop-iteration
    timestamp =timestamp
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
```

```
    done         =done
    ,@(per-arg-do arg-num :empty :bind)
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    ==>
    +imaginal>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         =done
    ,@(per-arg-do arg-num :fill
                  (if path-control-operator :fill :bind)
                  '=return-value nil "=arg~d" t
                  (when path-control-operator  arg-count)
                  (when path-control-operator '=arg0))
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    timestamp      =timestamp
    loop-iteration =loop-iteration
    last-argument ,(if path-control-operator (1- arg-count) arg-num)
    problem       =starting-order
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
                       ;branch-step-number 4
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :clear
    )

`(p ,(prod-name-fn "N-num-ok-commit")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value   =return-value
    dm-reload      :clear
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    ==>
    -imaginal>
    =retrieval>
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'done)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :empty
    )

`(p ,(prod-name-fn "N-num-fail")
    =goal>
    ISA metaproc
```

```
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value       =old
    - dm-reload    :reload
    ?retrieval>
    state  free
    state  error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@prod-slots-save-match-now
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =old
    dm-reload          :fail-commit
    )

`(p ,(prod-name-fn "N-num-fail-novalue")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value       :no-value
    - dm-reload    :reload
    ?retrieval>
    state  free
    - state error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@prod-slots-save-match-now
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       :no-value
    dm-reload          :fail-commit
    )

`(p ,(prod-name-fn "N-num-fail-save")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =next-return-branch
    next-branch-number =next-return-state
    return-value   =return-value
    dm-reload      :fail-commit
    starting-order =starting-order
    loop-iteration =loop-iteration
    timestamp =timestamp
```

```
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         t;=done
    ,@(per-arg-do arg-num :empty :bind)
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    ==>
    +imaginal>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         t
    ,@(per-arg-do arg-num :fill
                  (if path-control-operator :fill :bind)
                  '=return-value nil "=arg~d" t
                  (when path-control-operator  arg-count)
                  (when path-control-operator '=arg0))
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    loop-iteration =loop-iteration
    timestamp      =timestamp
    last-argument ,arg-num
    problem       =starting-order
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :fail-clear
    )

`(p ,(prod-name-fn "N-num-fail-commit")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value   =return-value
    dm-reload      :fail-clear
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    ==>
    -imaginal>
    =retrieval>
    =goal>
    current-branch     =branch
```

```
      branch-order       ,(prod-jump-fn 'done)
      operator           ,operator-sym
      return-value       =return-value
      dm-reload          :empty
      )

  )
 )
(:type-boolean
 (list
  `(p ,(prod-name-fn "B-all-subexpr-to-literal")
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order   ,(prod-jump-fn 'return)
      operator       ,operator-sym
      return-value   =return-value
      dm-reload      :reload
      subgoal        :empty
      loop-iteration =loop-iteration
      starting-order =starting-order
      timestamp      =timestamp
      ?retrieval>
      state free
      - state error
      ==>
      +retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order ,(prod-jump-fn 'dm-recall-parent t)
      op-name      ,operator-sym
      ,@prod-slots-save-match-now
      timestamp      ,(if (zerop arg-num) :no-value '=timestamp)
      last-argument  ,(if (zerop arg-num) :no-value (1- arg-num))
      loop-iteration ,(if (zerop arg-num) :no-value '=loop-iteration)
      =goal>
      return-value      =return-value
      current-branch    =branch
      branch-order      ,(prod-jump-fn 'return)
      operator          ,operator-sym
      dm-reload         :empty
      subgoal           :notgeneric
      )

  `(p ,(prod-name-fn "B-literal-from-subexpr")
      =goal>
      ISA metaproc
      current-branch =branch
      branch-order   ,(prod-jump-fn 'jump)
      operator      ,operator-sym
      - return-value  :empty
      - return-value  :no-value
      return-value    =return-value
      subgoal         :empty
      ?retrieval>
      state free
      - state error
      =retrieval>
      ISA op-sequence
      branch-name  =branch
      branch-order  ,(prod-jump-fn 'dm-recall-parent t)
      op-name       ,operator-sym
      return-branch =return-branch
      return-state  =return-state
      return-operator =return-op
      ==>
      =goal>
      return-value       =return-value
      branch-order       ,(prod-jump-fn 'return)
      operator           ,operator-sym
      current-branch     =branch
      subgoal            :notgeneric
```

```
    )
`(p ,(prod-name-fn "B-literal-from-subexpr-fallthrough")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'jump)
    operator       ,operator-sym
    return-value   :no-value
    subgoal        :empty
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym ;index-of-letter
    ,arg-slot        :empty
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-op
    ==>
    =retrieval>
    =goal>
    return-value       :no-value
    branch-order       ,(prod-jump-fn 'return)
    current-branch     =branch
    subgoal            :notgeneric
    )

`(p ,(prod-name-fn "B-value-ok")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    dm-reload      =dm-reload
    return-value   =return-value
    subgoal            :notgeneric
    ?retrieval>
    state free
    - state error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@(per-arg-do arg-num :empty :processed)
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :commit
    )

`(p ,(prod-name-fn "B-value-ok-save")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    return-value   =return-value
    subgoal            :notgeneric
    dm-reload      :commit
    starting-order =starting-order
```

```
        loop-iteration =loop-iteration
        timestamp =timestamp
        ?imaginal>
        state free
        ?retrieval>
        state free
        - state error
        =retrieval>
        ISA op-sequence
        branch-name  =branch
        branch-order ,(prod-jump-fn 'dm-recall-parent t)
        op-name      ,operator-sym
        done         =done
        ,@(per-arg-do arg-num :empty :bind)
        return-branch =return-branch
        return-state  =return-state
        return-operator =return-operator
        ==>
        +imaginal>
        ISA op-sequence
        branch-name  =branch
        branch-order ,(prod-jump-fn 'dm-recall-parent t)
        op-name      ,operator-sym
        done         =done
        ,@(per-arg-do arg-num :fill
                      (if path-control-operator :fill :bind)
                      '=return-value nil "=arg~d" t
                      (when path-control-operator  arg-count)
                      (when path-control-operator '=arg0))
        return-branch =return-branch
        return-state  =return-state
        return-operator =return-operator
        timestamp      =timestamp
        loop-iteration =loop-iteration
        last-argument ,(if path-control-operator (1- arg-count) arg-num)
        problem       =starting-order
        +retrieval>
        ISA op-sequence
        branch-name  =branch
        branch-order ,(prod-jump-fn 'dm-recall-parent t)
        op-name      ,operator-sym
        =goal>
        current-branch     =branch
        branch-order       ,(prod-jump-fn 'save)
        operator           ,operator-sym
        return-value       =return-value
        dm-reload          :clear
        )

    `(p ,(prod-name-fn "B-value-ok-commit")
        =goal>
        ISA metaproc
        current-branch =branch
        branch-order   ,(prod-jump-fn 'save)
        operator       ,operator-sym
        next-branch        =return-branch
        next-branch-number =return-state
        return-value   =return-value
        subgoal            :notgeneric
        dm-reload      :clear
        =retrieval>
        ISA op-sequence
        branch-name  =branch
        branch-order ,(prod-jump-fn 'dm-recall-parent t)
        op-name      ,operator-sym
        ?imaginal>
        state free
        ?retrieval>
        state free
        - state error
        ==>
        -imaginal>
```

```
    =retrieval>
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'done)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :empty
    subgoal            :empty
    )

`(p ,(prod-name-fn "B-value-fail")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value       =old
    - dm-reload    :reload
    subgoal            :notgeneric
    ?retrieval>
    state  free
    state  error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@prod-slots-save-match-now
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =old
    subgoal            :notgeneric
    dm-reload          :fail-commit
    )
`(p ,(prod-name-fn "B-value-fail-novalue")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'return)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
    return-value       :no-value
    - dm-reload    :reload
    subgoal            :notgeneric
    ?retrieval>
    state  free
    - state error
    ?imaginal>
    state free
    ==>
    -imaginal>
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    ,@prod-slots-save-match-now
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       :no-value
    subgoal            :notgeneric
```

```
    dm-reload          :fail-commit
    )

`(p ,(prod-name-fn "B-value-fail-save")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =next-return-branch
    next-branch-number =next-return-state
    return-value   =return-value
    subgoal            :notgeneric
    dm-reload      :fail-commit
    starting-order =starting-order
    loop-iteration =loop-iteration
    timestamp =timestamp
    ?imaginal>
    state free
    ?retrieval>
    state free
    - state error
    =retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         t
    ,@(per-arg-do arg-num :empty :bind)
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    ==>
    +imaginal>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    done         t
    ,@(per-arg-do arg-num :fill
                  (if path-control-operator :fill :bind)
                  '=return-value nil "=arg~d" t
                  (when path-control-operator  arg-count)
                  (when path-control-operator '=arg0))
    return-branch =return-branch
    return-state  =return-state
    return-operator =return-operator
    loop-iteration =loop-iteration
    timestamp      =timestamp
    last-argument ,arg-num
    problem       =starting-order
    +retrieval>
    ISA op-sequence
    branch-name  =branch
    branch-order ,(prod-jump-fn 'dm-recall-parent t)
    op-name      ,operator-sym
    =goal>
    current-branch     =branch
    branch-order       ,(prod-jump-fn 'save)
    operator           ,operator-sym
    return-value       =return-value
    dm-reload          :fail-clear
    )

`(p ,(prod-name-fn "B-value-fail-commit")
    =goal>
    ISA metaproc
    current-branch =branch
    branch-order   ,(prod-jump-fn 'save)
    operator       ,operator-sym
    next-branch        =return-branch
    next-branch-number =return-state
```

```
                    subgoal             :notgeneric
                    return-value    =return-value
                    dm-reload       :fail-clear
                    =retrieval>
                    ISA op-sequence
                    branch-name  =branch
                    branch-order ,(prod-jump-fn 'dm-recall-parent t)
                    op-name       ,operator-sym
                    ?imaginal>
                    state free
                    ?retrieval>
                    state free
                    - state error
                    ==>
                    -imaginal>
                    =retrieval>
                    =goal>
                    current-branch     =branch
                    branch-order       ,(prod-jump-fn 'done)
                    operator           ,operator-sym
                    return-value       =return-value
                    dm-reload          :empty
                    subgoal            :empty
                    )

                 )
                )
               (otherwise
                (list
                 `(p ,(prod-name-fn "ELSE-all-subexpr-to-literal")
                     =goal>
                     ISA metaproc
                     current-branch =branch
                     branch-order   ,(prod-jump-fn 'return)
                     operator       ,operator-sym
                     return-value   =return-value
                     dm-reload      :reload
                     subgoal        :empty
                     ?retrieval>
                     state free
                     - state error
                     ==>
                     +retrieval>
                     ISA op-sequence
                     branch-name  =branch
                     branch-order ,(prod-jump-fn 'dm-recall-parent t)
                     op-name       ,operator-sym
                     ,@prod-slots-save-match-now
                     =goal>
                     return-value      =return-value
                     current-branch    =branch
                     branch-order      ,(prod-jump-fn 'jump)
                     operator          ,operator-sym
                     dm-reload         :empty
                     )
                 )
                )
               )

            ))
         )

        ))

   debug-result
   )

)
```

Listing 54: ARGUMENT-P-SEQUENCE-FOR Code Listing