Production Systems and Rule-based Inference

Intermediate article

Gary Jones, University of Derby, Derby, UK
Frank E Ritter, Pennsylvania State University, University Park, Pennsylvania, USA

CONTENTS

Production systems are computer programs that reason using production rules. They have been used to create expert systems and models of human behavior.

REASONING AS RULE APPLICATION

A production system is a computer simulation of one or more tasks. Normally, the tasks have some form of goal to accomplish (for example, making a medical diagnosis based on symptom data). From a given starting position in the task the production system successively applies rules, each of which transforms the current task position, until a goal is reached. With certain sets of further constraints, production systems can be used to simulate how people perform tasks (particularly those of a problem-solving nature, which are well suited to the framework of production systems). Some examples of production system frameworks that have been used to simulate human behavior are Soar (Laird *et al.*, 1987), ACT-R (Anderson and Lebiere, 1998), and OPS5 (Forgy, 1981).

A production system consists of three components: a long-term memory (in the form of a rule base), a working memory, and an inference engine. The rule base contains rules, the conditions of which must be matched to elements in working memory. The inference engine determines which of the rules in the rule base have all their conditions matched to objects in working memory, and then decides which rule to apply. The application of a rule will usually cause elements in working memory to be added, removed, or altered. Further rules can then be matched by the inference engine.

What distinguishes a production system from any other system that simulates problem-solving behavior (such as a connectionist system) is its use of rules. A rule is an 'if–then' construct, where the 'if' part contains conditions that must be met in order for the rule to be considered for use. If the rule gets used ('fired' or 'applied'), the actions in the 'then' part are performed. The conditions are sometimes called the left-hand side (LHS) of the rule, and the actions the right-hand side (RHS).

Production system behavior can be explained using an example problem of moving a sphere to the exit of a room. Figure 1 shows the layout of a room as a seven-by-eight grid where black squares represent obstacles. Given this scenario there are a variety of rules that will help in moving the sphere to the exit, such as:

```
IF the exit is above the sphere AND there
  isn't an obstacle directly above the sphere
  THEN move the sphere upwards one
  space.                                    (1)
```

There are two conditions in rule 1 that need to be met before the action part of the rule is applied: the exit position must be above the sphere, and the position directly above the sphere must be free to

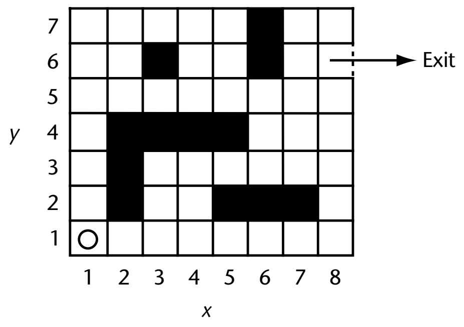

Figure 1. Problem scenario where a sphere has to be guided to the exit of a grid.

move into (i.e. it is not obstructed). If these two conditions apply to the current situation, then the sphere will be moved upwards one space.

Rules are often called productions, or production rules, because they necessarily produce something when they get used (normally changing something within working memory).

At any one time, there may be more than one rule whose conditions are satisfied. The process of collecting all applicable rules into a 'conflict set', and then selecting one of the rules in the conflict set to be fired, is called the recognize–act cycle. The production system stops either when no more rules are able to fire (e.g. ACT-R) or when the task is known to be complete (e.g. Soar).

In addition to being at the core of many expert systems, production systems are often used to simulate human behavior (e.g. Kieras and Polson, 1985; Klahr *et al.*, 1987; Newell, 1990; Newell and Simon, 1972; Young, 1976). By successfully simulating human behavior on a task, the production system suggests the processes that may be used when humans perform the task. Although some researchers believe that human thinking need not be rule-based (e.g. Rumelhart *et al.*, 1986; Brooks, 1991), there are many (such as Langley, Anderson, and Newell) who believe that it is, or at least can be, fruitfully viewed that way. The question revolves around whether or not human thinking deals with symbols. Rule-based systems use symbols, whereas others, such as connectionist systems, do not. The 'symbolic versus subsymbolic' question is not covered here, but is still widely debated within the cognitive science community.

COMPONENTS AND MECHANISMS OF PRODUCTION SYSTEMS

Two of the three components of a production system are distinct types of memory: working memory (facts) and long-term memory (the rule base). The third component is the inference engine.

Working memory usually contains facts about the world that are relevant to the task the production system is performing. Working memory is usually represented by attribute–value pairs. One element in working memory can have several attribute–value pairs. Using the sphere example, one element (or fact) in working memory will be the position of the exit of the grid. This element might have two attribute–value pairs: the first attribute will be '*x*-coordinate', having the value '8', and the second attribute will be '*y*-coordinate' having the value '6'.

Long-term memory usually consists of rules that govern the behavior of the production system. The inference engine combines working memory and long-term memory by finding rules whose conditions are matched by elements in working memory. When this happens, the rule can fire.

Production systems work in a cycle of production firings (the recognize–act cycle). Normally, only one production is fired on each cycle. The action of the production often changes what is in working memory and thus enables another production rule to fire. The resulting change may in turn enable a further production rule to fire. This is how the production system produces behavior (for example, maneuvering the sphere to the exit). When no further production rules can be fired, the system halts.

Let us work through an example illustrating this cycle and showing how production application works. Given the problem scenario in Figure 1, we noted that rule 1 matched working memory and could be applied. In that rule, there is a condition that specifies that the exit should be above the sphere's current position. If we did not already know that this was the case, then we would need a rule to work out whether or not the exit was above the sphere. The following rule could accomplish this:

```
IF the y-coordinate of the exit is greater
  than the y-coordinate of the sphere THEN
  put in working memory that the exit is
  above the sphere.                        (2)
```

Similarly we could have a rule to determine whether there was an obstacle directly above the sphere:

```
IF there is no obstacle having a y-coordinate
  of 1 less than the y-coordinate of the
  sphere THEN put in working memory
  that there isn't an obstacle directly above
  the sphere.                              (3)
```

Now suppose the starting position of the room is as shown in Figure 1. Certain elements would be in working memory, for example, the position of the sphere, the position of the exit, and the positions of obstacles.

Rules 2 and 3 can now be matched. The *y*-coordinate of the exit can be compared with the *y*-coordinate of the sphere; and, as it is in fact greater, rule 2 can fire. This places the element 'exit is above the sphere' in working memory. Rule 3 can also fire: all the obstacles can be checked as to whether they are directly above the sphere; as none are, the element 'no obstacle directly above sphere' is placed in working memory. Note that these two rules could have fired at the same time, but most production systems would fire them in sequence (i.e. one recognize–act cycle for each). We will see later, when considering conflict resolution, how the production system determines which rule to fire first.

After these two new elements have been placed in working memory, rule 1 can fire, which moves the sphere upwards one space. Figure 2 shows how working memory changes during the three recognize–act cycles.

INFERENCE THROUGH FORWARD AND BACKWARD CHAINING

The normal way in which a production system processes rules is by matching elements in working memory in the 'if' part of the rule, and then applying the actions in the 'then' part of the rule. This is how the production system in the example above has cycled.

The production system can be seen as working forwards from the starting position (e.g. the problem position in Figure 1) towards a finishing position (the goal), in a series (a chain) of rule applications. This is called forward chaining or data-driven reasoning. Forward chaining is used in the majority of production system domains, including cognitive models of problem solving (an instructive example is the Tower of Hanoi (Anzai and Simon, 1979)).

The opposite of forward chaining is to match the actions of rules to working memory, and if all can be matched, to apply the conditional part of the rule. This method works backwards from the goal to the starting position (i.e. the initial data) and is therefore known as backward chaining, or goal-driven

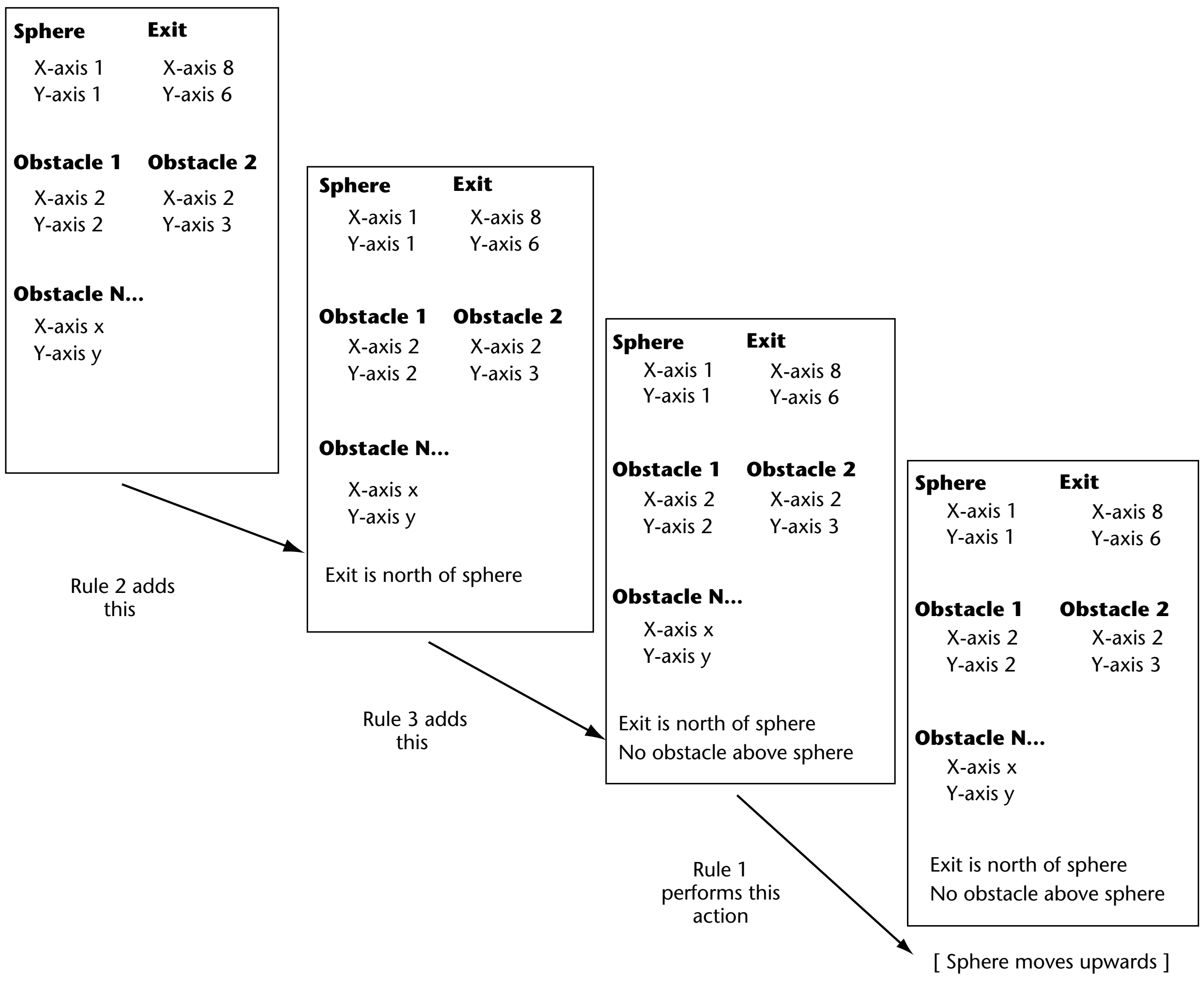

Figure 2. How working memory changes as the sphere production system rules fire.

reasoning. Backward chaining is normally used in domains where you wish to reason backwards from the final state of the world, such as discovering what preconditions led to a patient showing their current set of symptoms. The 'actions' of rules in these cases are often either changes to working memory or questions (to the user of the system) which need to be answered in order to add more facts into working memory. Expert systems often incorporate backward chaining so that they are able to explain how they arrived at their decisions.

Rules and knowledge can be represented at multiple levels. For example, Able (Larkin *et al.*, 1980) represents how learning in formal domains such as physics shifts novices from a backward chaining approach of trying to find the values of variables in problem solving, to a forward chaining approach of experts where unknown variables are simply and directly derived from known variables. This model has been implemented in Soar (Ritter *et al.*, 1998), which is normally seen as a forward chaining system because it applies its production rules towards a goal. In this case, the domain knowldge rules are implemented as sets of Soar rules. The domain knowledge is initially applied in a backward chaining way, searching from the target variables of the physics problem back towards the givens. With learning, the direction of processing on the knowledge level reverses.

CONFLICT RESOLUTION AND PARALLEL PRODUCTION FIRING

There are occasions when more than one rule can be fired for a given set of working memory elements. In our example above, rules 2 and 3 could both be matched. Figure 3 shows how these two rules could be matched for the elements in working memory that we started with in the example. The general approach in production systems is for the matching process to occur in parallel but the fining process to occur in serial. In the matching process, the inference engine determines which rules have all their conditions matched by elements in working memory. The resulting set of all rules that can be applied is called the conflict set.

When it is possible to fire more than one rule for a given situation, the production system is said to be in conflict. Firing all of the rules in the conflict set in parallel can give rise to inconsistent knowledge and results. Production systems generally resolve this problem by selecting only one of the rules to fire. The selection is made by the inference engine, using 'conflict resolution'. Production systems have used a variety of approaches including:

Textual order. This is the simplest resolution of conflict: simply choose the rule that comes first in the rule base.

Refractoriness. The same rule cannot be applied to the same working memory elements more than once. The inference engine needs to keep track of when elements in working memory were added or changed, in order to calculate whether a rule is being applied on exactly the same elements of working memory or whether there has been a change to those elements.

Recency. Apply the rule whose conditions match the most recently added working memory elements. This technique encourages adaptivity.

Specificity. Choose the rule that is either the most specific (i.e. has the largest number of conditions) or is the least specific (i.e. has the smallest number of conditions). Which of these is chosen is dependent upon the type of domain that is being

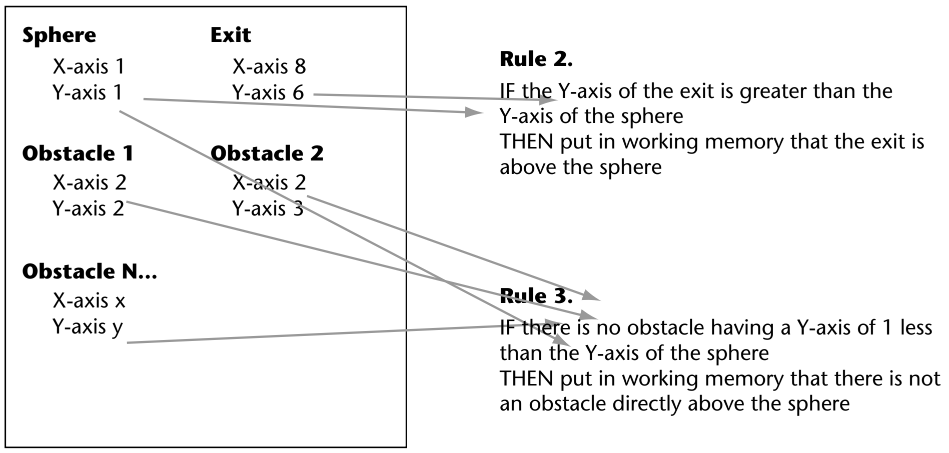
Sphere
X-axis 1
Y-axis 1
Exit
X-axis 8
Y-axis 6
Obstacle 1
X-axis 2
Y-axis 2
Obstacle 2
X-axis 2
Y-axis 3
Obstacle N...
X-axis x
Y-axis y
Rule 2.
IF the Y-axis of the exit is greater than the Y-axis of the sphere
THEN put in working memory that the exit is above the sphere
Rule 3.
IF there is no obstacle having a Y-axis of 1 less than the Y-axis of the sphere
THEN put in working memory that there is not an obstacle directly above the sphere

Figure 3. How working memory elements can be matched to conditions in rules.

modeled. For example, in a medical domain we may wish to be as certain as we can when applying a rule, and so we may set the most specific rule to be applied; in a domain involving search, we may wish to be less specific so that the system is less likely to arrive at a dead end.

Saliency. This allows the person writing the production system to set (numerically) how important each rule is. The rule with the highest saliency is selected. Conflict resolution is therefore primarily the responsibility of the production system designer.

Meta-rules. This allows the set of rules that are in conflict to be pruned or reordered based on a higher-order rule. For example, if the conflict set contains two rules, one which mentions high blood pressure leading to a possible heart condition and the other mentioning high blood pressure leading to a possible high temperature, then the production system designer could add a meta-rule which stated that if these two rules are in conflict, then choose the former over the latter.

Although these techniques are usually used independently of each other, there are occasions when more than one needs to be used. For example, it would be possible, when using refractoriness, to be unable to select a unique rule (for example, at the beginning of a production system run). Often, more than one type of conflict resolution is used, but one type is given priority.

Soar has taken a different approach. All matched rules are allowed to fire in parallel, but they are not allowed to modify working memory directly. They provide suggestions for changes to working memory, and a preference calculus is used to resolve contradictions and implement the changes.

STRUCTURE OF PRODUCTIONS

Productions can be represented in numerous ways. They can be represented as plain sentences (like rules 1–3 above); they can be represented in a structured editor as objects; or they can be represented as a list of clauses, which is how Soar and ACT-R represent them. The code sample below shows how rule 2 could be represented in ACT-R:

```
(p check-if-exit-is-above-sphere
  =goal>
    ISA           move-sphere-to-exit
    to-move       =sphere
    exit          =exit
  =exit>
    ISA           Exit
    Y-coordinate  =exit-y-coordinate
  =sphere>
    ISA           Sphere
    Y-coordinate  =sphere-y-coordinate
  =greater-than-fact>
    ISA           Greater-than-fact
    big-num       =exit-y-coordinate
    small-num     =sphere-y-coordinate
==>
  =new-working-memory-element>
    ISA           Sphere-location-fact
    location      exit-is-above-sphere
)
```

Parentheses are used to delineate the rule, which is introduced by `p` and then a name. This rule has four conditions, each consisting of a single working memory element; these are marked by = and then a name (in ACT-R – other systems use different conventions). Each condition must be matched to elements that are present in the working memory of the system.

Every element in working memory in ACT-R has an `ISA` ('is a') attribute–value pair, which defines the working memory element type. All variables in ACT-R are preceded by =. The first condition of a rule in ACT-R is always the `goal` condition. (The goal is often part of the conditions of rules in goal-directed production systems.) The `goal` condition states that there must be a goal in working memory of the type `move-sphere-to-exit`. If this does exist, it is matched; the value of the `to-move` attribute is placed in the `=sphere` variable, and the value of the `exit` attribute is placed in the `=exit` variable. Note how these same variables are then used in the next two conditions, signifying that what is `to-move` should be a memory element of the type `Sphere`, and where it is moved to should be a memory element of the type `Exit`.

The second condition specifies a match to an `Exit` element in working memory, and if this element exists it must have a `Y-coordinate` value (this will be stored in the `=exit-y-coordinate` variable). The third condition specifies a match to a `Sphere` element in working memory, and if this element exists it must also have a `Y-coordinate` value (this will be stored in the `=sphere-y-coordinate` variable). The fourth condition specifies a match to a `Greater-than-fact` element in working memory. This denotes knowledge of which numbers are larger than others. The variables that were set in the second and third conditions are used in the fourth condition to see if `=exit-y-coordinate` is greater than `=sphere-y-coordinate`. If the rule can be fired, then a new working memory element is created, which specifies that the exit is above the sphere.

The number of input or output clauses is not limited by this syntax. In ACT-R the variables are bound (i.e. matched) in the order in which they are written, whereas in Soar they need not be. Rules in ACT-R and some other systems also have weights associated with them. These weights are updated by learning algorithms to represent the rule's probability of success, its cost, and other attributes. The weights may be used in a variety of ways, for example, to choose the most useful rule. Soar does not include these attributes.

THE RETE ALGORITHM

Most production systems include hundreds of rules; some include thousands. One includes nearly a million rules (Doorenbos, 1994). If each rule is checked individually to see if it matches, the time taken to create the conflict set will depend linearly on the number of clauses in the whole rule set. With small rule sets, this is not a problem; but as the size of models in production systems has grown, this causes a significant bottle-neck, particularly where the clauses have to be matched against sets of objects.

The RETE algorithm (Forgy, 1982) was created as a way to speed up the matching process by taking advantage of the reuse of clauses and the fact that working memory elements change slowly. For example, consider the three rules below, based on the sphere example:

IF the exit is above the sphere's current position AND there isn't an obstacle directly above the sphere THEN move the sphere upwards one space. (4)

IF the exit is above the sphere's current position THEN move the sphere upwards one space. (5)

IF there isn't an obstacle directly above the sphere THEN move the sphere upwards one space. (6)

These rules overlap, both in their conditions and in their actions. (The example is for illustration only: it would be unwise to move the sphere upward by only checking if the exit was above it, as rule 5 does.)

In essence, the RETE algorithm creates a network representing the whole rule set. The matching is then based on changes to working memory. As elements leave or enter, the state of the network is updated. Thus the time taken by the matching process depends linearly on the number of changes to working memory instead of on the number of rules.

For example, the RETE network for rules 4–6 would combine the first clauses of rules 4 and 5 as a top node. If the exit's position changed, then and only then would the clause be updated. When the conflict set is needed, all the clauses that are matched are already noted in the network. So if the exit is above the sphere's current location, then rule 5 would be waiting in the conflict set. Whether rule 4 was in the conflict set would depend on whether the working memory element matching its second clause had been added. The addition of the working memory element would also have satisfied rule 6 at the same time.

The presence of the RETE algorithm is not always noticed by people who use production systems, but it has drastically improved their speed and therefore their usefulness. RETE is used by both OPS5 and Soar. Further refinements of match optimization have also been developed.

PRODUCTION SYSTEMS AS COGNITIVE ARCHITECTURES

A cognitive architecture proposes a theory of the human information processing apparatus. Some production systems, such as ACT-R and Soar, have been used to implement such theories. Their intention is to model the types of process and structures that generate human behavior (e.g. what the constraints on memory are), and use these models to simulate human behavior. If the same architecture can be used to accurately simulate behavior across domains, then this provides evidence that the human brain may resemble the cognitive architecture. Cognitive architectures have been used to simulate the behavior of both adults (e.g. Jones *et al.*, 1999) and children (e.g. Jones *et al.*, 2000).

SUMMARY

Productions systems have been used to organize and apply knowledge in a variety of domains. While there remain questions as to whether knowledge can be modeled directly as structures in a production system, they provide a fruitful way to think about human behavior.

References

Anderson JR and Lebiere C (1998) *The Atomic Components of Thought*. Mahwah, NJ: Erlbaum.

Anzai Y and Simon HA (1979) The theory of learning by doing. *Psychological Review* **86**: 124–140.

Brooks RA (1991) Intelligence without representation. *Artificial Intelligence* **47**: 139–159.
Doorenbos RB (1994) Combining left and right unlinking for matching a large number of learned rules. In: *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. Menlo Park, CA: AAAI Press.
Forgy CL (1981) *OPS5 User's Manual*. [Technical Report CMU-CS-81–135, Department of Computer Science, Carnegie-Mellon University, Pittsbugh, PA.]
Forgy CL (1982) Rete: a fast algorithm for the many pattern / many object pattern match-problem. *Artificial Intelligence* **19**: 17–37.
Jones G, Ritter FE and Wood DJ (2000) Using a cognitive architecture to examine what develops. *Psychological Science* **11**: 93–100.
Jones RM, Laird JE, Nielsen PE *et al*. (1999) Automated intelligent pilots for combat flight simulation. *AI Magazine* **20**: 27–41.
Kieras D and Polson PG (1985) An approach to the formal analysis of user complexity. *International Journal of Man–Machine Studies* **22**: 365–394.
Klahr D, Langley P and Neches R (eds) (1987) *Production System Models of Learning and Development*. Cambridge, MA: MIT Press.
Laird JE, Newell A and Rosenbloom PS (1987) Soar: an architecture for general intelligence. *Artificial Intelligence* **33**: 1–64.
Larkin JH, McDermott J, Simon DP and Simon HA (1980) Models of competence in solving physics problems. *Cognitive Science* **4**: 317–345.
Newell A (1990) *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press.
Newell A and Simon HA (1972) *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.
Ritter FE, Jones RM and Baxter GD (1998) Reusable models and graphical interfaces: realising the potential of a unified theory of cognition. In: Schmid U, Krems J and Wysotzki F (eds) *Mind Modeling: A Cognitive Science Approach to Reasoning, Learning and Discovery*, pp. 83–109. Lengerich, Germany: Pabst Scientific Publishing.
Rumelhart DE, McClelland JL and the PDP Research Group (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. I 'Foundations'. Cambridge, MA: MIT Press.
Young RM (1976) *Seriation by Children: An Artificial Intelligence Analysis of a Piagetian Task.* Basel: Birkhauser.

Production–Comprehension Interface

Intermediate article

Victor S Ferreira, University of California, San Diego, California, USA

CONTENTS

Many features of spoken language, from the nature of child-directed speech to audience design effects, reflect a sensitivity in language production to the needs and strategies of language comprehension. These adjustments, sometimes deliberate, sometimes automatic, ensure successful communication.

INTRODUCTION

The primary reason that speakers speak is so that their listeners can understand them. It is therefore unsurprising that unlike many cognitive tasks (such as perceiving, remembering, or decision-making), the language production performance of an individual speaker must take into account the processing capabilities and the knowledge states of another, namely, that speaker's intended listener. Research into this topic of the nature of the production–comprehension interface generally explores how production accommodates its processing to the needs of comprehension.

This article discusses the production–comprehension interface by exploring two related issues. The first concerns how speakers cater specific details of their utterances to take into account the knowledge and the comprehension capabilities and