Chapter 1.4

Human Cognition and Programming

Tom Ormerod

Department of Human Sciences, Loughborough University of Technology, Loughborough, Leicestershire LE11 3TU, UK

Abstract

This chapter presents an overview of cognition, and reviews the factors that dictate the nature of cognitive models of programming. The computational metaphor is outlined, and the following issues central to information processing are described: knowledge representation, schemas, production rules, procedural and declarative knowledge, attentional and memory resources, semantic memory, problem solving, skill acquisition, and mental models. These issues are fundamental to psychological models of programming presented in later chapters.

1 The relationship between cognition and programming

The fields of cognition and programming are related in three main ways. First, cognitive psychology is based on a 'computational metaphor', in which the human mind is seen as a kind of information processor similar to a computer. Secondly, cognitive psychology offers methods for examining the processes underlying performance in computing tasks. Thirdly, programming is a well-defined task, and there are an increasing number of programmers, which makes it an ideal task in which to study

Psychology of Programming
ISBN 0-12-350772-3

cognitive processes in a real-world domain. This chapter presents an overview of cognition, and reviews the factors which constrain cognitive performance and which dictate the nature of cognitive models of programming.

1.1 The computational metaphor

Cognitive psychology is the study of the mechanisms by which mental processes are carried out, and the kinds of knowledge required for each process. Cognitive processes include perception, attention, memory storage and retrieval, language production and understanding, problem solving and reasoning. These are all operations performed on information by a central processing unit (CPU) with associated memory, to produce an appropriate output. Like a computer, information flows through stages of cognitive processing and storage to give a response output. Also like a computer, information is represented in symbolic form. Recent theories of information processing retain the concept of information in the form of symbols being stored and operated upon before output of a response, although their components are more interactive and less stage dependent than those of earlier theories (e.g. Broadbent, 1958).

Pylyshyn (1984) argues that, since both human and computer output result from operations carried out on symbols, cognition is literally a kind of computation. A computer system can be described at three levels: the software; features of the implementation of programs on the hardware (e.g. the disk-operating system, CPU speed, RAM cache, etc.); and the hardware itself (e.g. circuit boards, keyboard input, etc.). Adopting a literal interpretation of the computational metaphor, similar levels of software, implementation and hardware can be used to describe a cognitive system (e.g. Anderson, 1987a). The 'cognitive software' consists of the mental procedures and knowledge representations used in performing cognitive tasks. For example, strategies such as means-ends-analysis are equivalent to programs for undertaking certain problem-solving tasks. The 'cognitive implementation' of software concerns the mechanisms for carrying out mental procedures and knowledge representation, such as symbol manipulation, storage, retrieval and so on. For example, limitations of attention and memory constrain the knowledge and procedures available during problem solving. Anderson (1987a) argues that studying cognitive implementation, despite identifying constraints on cognitive performance, is both more difficult and less rewarding than studying cognitive software. Although this is contentious, it is consistent with research into programming which is mainly concerned with identifying the mental procedures and knowledge used in programming.

The 'cognitive hardware' consists of the physiological structures, notably the brain, upon which cognitive processes are implemented. Physiological structures are not a central concern of cognitive psychology, since one of the major assumptions underlying the computational metaphor is that cognitive processes are determined by their function, and not by the structure of the hardware upon which they are based. The computational metaphor dominates cognitive psychology, although some argue that accounting for cognition in terms of symbol manipulation is implausible, and that the biological basis of cognition should not be ignored (e.g. Searle, 1980). A recent development, which offers an alternative to the computational metaphor for studying cognition, is 'connectionism' (e.g. Rumelhart and McClelland, 1986). Connectionist modelling is closer to the level of cognitive hardware, in that it uses the parallel processing capabilities of the brain to explain how cognitive processes are carried out. Connectionist models are not of direct relevance to programming research because they focus on cognitive hardware rather than software. However, future developments in parallel computer architectures may include concepts such as constraint satisfaction programming (see Chapter 1.2). The cognitive skills necessary to cope with programming a parallel-processing machine may therefore be an important topic for future research.

1.2 Important themes in cognition

A division is often made between knowledge representation and the processes which operate on representations of knowledge. At the level of cognitive implementation, theories of representation describe the form in which individual units of knowledge are stored. Pylyshyn (1979) claims that all information is represented as propositions, which are language-like assertions retaining only the meaning of individual items of knowledge, rather than representing knowledge in the exact form in which it arrives as perceptual input. Alternatively, Paivio's (1971) 'dual code' theory suggests that verbal information is represented in linguistic form, and visual information is represented in spatial image or analogue form. Analogue representations may determine performance in some programming tasks. For example, Ormerod *et al.* (1986) found that different reasoning strategies were used in comprehending diagrammatic and list representations of Prolog clauses. More important to programming research, however, is the organization of knowledge at the level of cognitive software (for example, the schema and strategy accounts of programming knowledge discussed in Chapters 3.1 and 3.2).

1.2.1 Schema representations

An important approach to describing knowledge representation at a cognitive software level is the 'schema'. A schema (after Bartlett, 1932) consists of a set of propositions that are organized by their semantic content. A perceptual input, cognitive goal or output of a cognitive process may evoke a schema with a related semantic content. Once evoked, it provides an organized body of knowledge which is appropriate for the task in hand. For example, Figure 1 shows a schematic representation of part of the knowledge a programmer may possess about programming languages. If a programmer is required to think of an alternative to iteration for dealing with lists, the words 'iteration' and 'lists' will activate the schema 'programming languages'. This will allow the programmer to access knowledge about looping constructs, and thus generate alternatives such as 'recursion'. There are two basic principles of schema theories: first, that cognitive processing is guided and limited by the application of prior knowledge; and secondly, that schemas contain relatively abstract knowledge which is independent of any one event. Thus, a schema highlights relevant information in a task domain, and may add new information when there is insufficient information in a task domain. Figure 1 illustrates a number of components of schemas, such as hierarchical organization, default values for generating information which is not present in the task domain, and slots which can be filled by matching information from the task domain. A schema offers a method for limiting the amount of inputed information, or 'bottom-up' control, that is needed to perform a task. Schemas provide 'top-down' control, using prior knowledge to restrict the operations that may be undertaken.

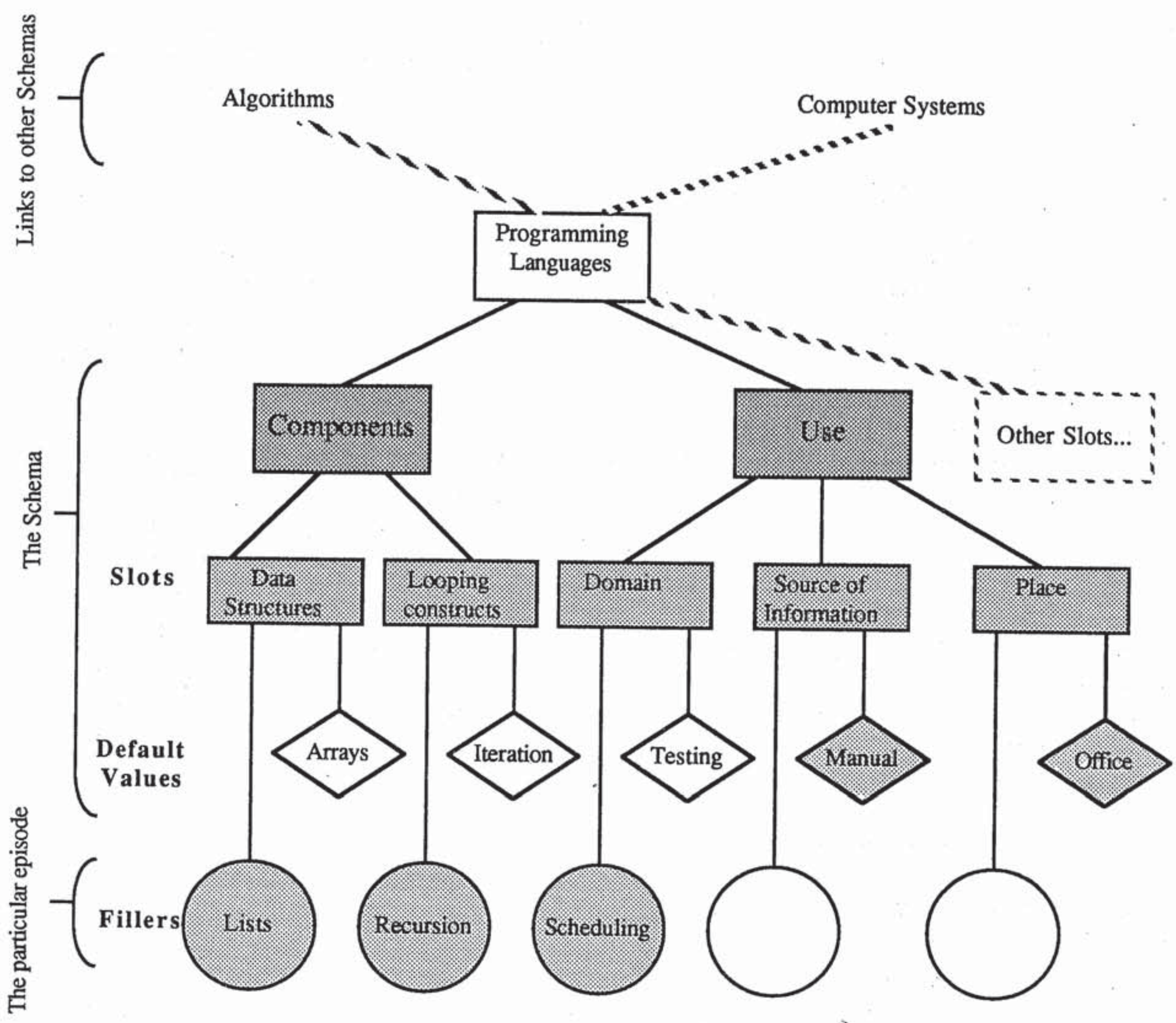

Figure 1: The schema 'programming languages' represents part of the general knowledge a person may have about programming languages. This schema is linked to others (e.g. 'algorithms'). The particular episode represents an encounter with a previously unknown language (such as Lisp). The shaded areas represent information available for a particular task. Information may originate from the task domain as 'fillers' for the schema slots, or from the schema as 'default values' where no conflicting task domain information is present. This representation of schemas, based on one devised by Cohen *et al.* (1986, p. 28), is only one of many possible representations. For example, the 'script' shown in Figure 3. is also a schematic representation.

Schema theories appear in various guises, such as 'frames' (Minsky, 1975), which are schemas originally employed in artificial intelligence (AI) models of vision. Schank and Abelson (1977) identify a particular kind of schema known as the 'script'. A script is a sequence of abstracted actions which occur in common events, with slots for specific instances. For example, a 'restaurant' script contains a sequence of general scenes such as entering, seating, ordering, paying and leaving, along with specific exceptions (e.g. the time someone shouted for service). Schank and Abelson (1977) also identify 'plans', which determine the inferences required for understanding situations for which there are no stereotypical event sequences. Scripts describe action sequences which have occurred many times (e.g. visiting a restaurant), whereas plans describe the production of novel action sequences (e.g. robbing a liquor store). The inferences determined by a script are based on prior knowledge, whereas plan inferences are based on achieving goals in the absence of task-specific knowledge. The processes involved in 'planning' have been described in a number of AI models (e.g. Wilensky, 1981), though there have been surprisingly few empirical investigations of planning. Somewhat confusingly, the programming plans (Soloway and Erlich, 1984) discussed in Chapter 3.2 are closer to scripts of programming knowledge.

Schema theories have been used to account for a large number of cognitive processes, such as memory storage and retrieval (e.g. Rumelhart and Norman, 1983), language understanding (e.g. Schank and Abelson, 1977) and problem solving (e.g. Gick and Holyoak, 1983). There is a danger of the term 'schema' being too nebulous to be of use, and there is a tendency to use the terms 'schema' and 'knowledge' interchangeably. Therefore, it is important for schema theories of programming knowledge to specify explicitly the mechanisms which mediate the acquisition and use of knowledge. Also, a schema theory must be descriptive rather than prescriptive, In other words, it must model the knowledge that programmers really use rather than the knowledge they ought to use.

1.2.2 Production rule representations

Another representation is the 'production rule'. Production rules emphasize the process aspects of cognition, whereas schemas emphasize the representational aspects. A production rule consists of two propositions forming a 'condition-action' pair, one of which is a goal or desired cognitive state, the other being the action or subgoals required to achieve that state. Figure 2 shows examples of some production rules used in Lisp programming (Anderson *et al.*, 1984). If the conditions of a production rule are satisfied by matching them with a perceptual input, retrieval of knowledge from a long-term store, or by the action of another rule succeeding, then the rule is fired or 'activated' (i.e. made accessible to conscious attention) and its action is undertaken. The successful firing of a production rule represents a cognitive step. As an example from Figure 2, a novice learning Lisp by solving textbook exercises might use the production rule P1 as the first step in writing a new function. The conditions would be matched by the existence of a goal to write a function and the prescence of a previous example in the textbook, and the actions to compare the example to the required function and then map the example's solution onto the present problem would be carried out. Production rules are described by Anderson (1987a) as a programming language for cognitive software, in that sets of rules combine to give a 'production system' for undertaking a cognitive task, where the action of one rule provides the conditions for firing the next rule. Production systems have been

A general production rule used by novices in the declarative stage

```
P1:  IF the goal is to write a function and there is a previous example
     THEN set as subgoals
         1. to compare the example to the function
         2. map the example's solution onto the current problem.
```

....

Compilation of a production rule for list processing

Proceduralization

```
P2:  IF the goal is to code a relation defined in an argument
         and there is a Lisp function that codes this relation
     THEN use function with the argument and set as a subgoal to code the argument.
```

Becomes;

```
P3:  IF the goal is to code the first member of a list
     THEN use CAR of the list and set as a subgoal to code the list.
```

Composition

Given another production rule P4;

```
P4:  IF the goal is to add an element to a list
     THEN use CONS on the element and the list
         and set as subgoals to code the element and to code the list
```

Adding P3 to P4 gives;

```
P5:  IF the goal is to add the first member of one list to another list
     THEN CONS the CAR of the first list to the second list
         and set as subgoals to code the first list and to code the second list
```

....

A specific production rule for experts in the procedural stage

```
P6:  IF the goal is to check that a recursive call to a function will terminate
         and the recursive call is in the context of a MAP function
     THEN set as a subgoal to establish that the list provided to the MAP function
         will always become NIL after some number of recursive calls.
```

Figure 2: Some possible production rules underlying Lisp programming skills at different stages of skill acquisition, based on Anderson *et al.* (1984). In each production rules the 'IF ...' antecedents give the conditions for the rule to be fired, and the 'THEN ..' consequents give the actions that occur if the consequents are met.

proposed to underlie many cognitive processes, notably those involved in problem solving (e.g. Anzai and Simon, 1979).

Production rules are frequently used to represent 'procedural' knowledge, that is knowledge about *how* to carry out a cognitive task, whereas schemas are often used to represent 'declarative' knowledge, that is knowledge about *what* constitutes a task domain. This parallels a distinction (described in Chapter 1.1) between declarative languages (e.g. Prolog) and procedural languages (e.g. Pascal). It has been claimed that a declarative style is more natural than a procedural style of knowledge representation, and hence that declarative programming languages have advantages over procedural languages (e.g. Miller, 1974). However, others suggest that the preferred style is context dependent (e.g. Rumelhart and Norman, 1983; Gallotti and Ganong, 1985). A related issue is the extent to which a person has conscious access to their knowledge. Whilst declarative knowledge is available to conscious inspection and control, procedural knowledge is applied automatically without conscious control. Again, the distinction is not absolute, and depends on the task and level of expertise involved. The issue of control over cognitive processing has a number of implications for programming, and is discussed in the next section.

2 Constraints on cognitive skills

Programming is a 'high-level' cognitive task, in that it involves problem solving and linguistic skills which require a number of 'lower-level' cognitive tasks to be carried out at the same time. These include the perception, attention to relevant aspects, and short-term storage of task information, and the retrieval of relevant long-term knowledge. Performance in these tasks dictates a number of constraints on programming performance.

2.1 Resources for cognitive processing

Increasingly, theories viewing attention and short-term memory as isolated operations which restrict perceptual input (e.g. Broadbent, 1958) have been replaced by theories of limited 'resources' which restrict both input and output. For example, early theories of memory (e.g. Atkinson and Shiffrin, 1968) identified a passive short-term store with a limited storage capacity of approximately seven items (Miller, 1956), and a long-term store of unlimited capacity. Recent accounts maintain the distinction between short- and long-term stores, but suggest a more active role for short-term memory. Baddeley and Hitch (1974) propose a 'working memory' model of short-term memory. The main component is a central executive, which allots limited resources to either storage or processing. The central executive is served by sub-systems for rehearsal of verbal and spatial information. A working memory system is embodied in a number of cognitive theories (e.g. Anderson, 1983).

The exact nature of cognitive resources is unspecified, but the term implies a limited availability of conscious effort and storage capacity. The performance of two demanding tasks at the same time is difficult, not because information from only one task is available, but because the cognitive system lacks the resources to perform both tasks. Since the number of processes involved in programming is large, attentional and memory resources must be divided up amongst competing processes, thereby limiting performance. Resource theories may explain the source of some programming errors. For example, insufficient short-term memory resources have been suggested to account for errors made by novice Lisp programmers (Anderson and Jeffries, 1985). A resource account of programming errors implies that performance can be improved if the resource requirements of a task are decreased. For example, using appropriate perceptual cues in notations may reduce the attentional demands of program comprehension (this is discussed further in Chapter 2.2).

2.1.1 Controlled and automatic processes in attention

Shiffrin and Schneider (1977) propose a theory of 'controlled and automatic processes' in attention. Automatic processes are analogous to compiled programs in that they are carried out directly in terms of the cognitive implementation or hardware, whereas controlled processes are like interpreted program code that requires translation into machine-specific terms at run-time. Controlled processes require attentional resources, are of limited capacity, and can be used in different contexts. Automatic processes do not require attention, are not capacity limited, but like compiled programs are not modifiable. For example, an expert Lisp programmer may write the syntax of a function definition as an automatic process, but may have to use controlled processes to specify the function itself. In the same way that compiling code speeds up the running of a program, repeated practice reduces the attentional demands of a task by allowing previously controlled processes to be automated.

The activation of automatic processes precludes conscious access to cognitive processing, thereby preventing the monitoring of task performance. As a consequence, the inappropriate activation of automatic processes may lead to errors in performance (Reason, 1979). Automated knowledge of programming skills releases resources which can be used for other programming tasks. However, automatic processes are restricted in their application, and their use in programming tasks which differ from the learning domain may lead to errors. For example, a common error made by Prolog novices who are experienced in Lisp programming is to add unnecessary brackets at the beginning of Prolog program clauses. This kind of error is minor if there are sufficient resources for the controlled processes required for debugging, but it becomes more significant when resources are tied up by other processing demands.

2.2 The contents of memory

A number of factors affect the encoding, storage, and retrieval of long-term memories (for a review, see Eysenck, 1984). Encoding is affected by the depth of processing taking place as information is presented, as well as the distinctiveness of the information to be encoded. Storage is affected by the nature of information rehearsal, elaborative rehearsal being more effective than maintenance rehearsal. Retrieval of information has been shown to be a function of the cues available both at encoding and retrieval time, and the nature of the task (e.g. whether recognition or recall is required). A distinction is often made between semantic memories which contain general knowledge about the world, and episodic memories which contain personally encountered events and experiences linked to specific times and places. A strong division between the two memory systems (e.g. Tulving, 1985) is contentious. In programming research, usually only semantic memory is examined, although structures in semantic memory must be constructed from individual episodes. This is important in learning programming, where schematic knowledge of programming concepts is developed by generalizing across a number of repeated learning episodes.

2.2.1 Semantic networks

One of the most important representational systems proposed for long-term memory is the 'semantic network' (Collins and Quillian, 1970). In a semantic network, semantically related propositions are represented as nodes linked together in a hierarchical fashion. So, in a network of semantic knowledge about animals, 'animal' would be above 'bird', which would be above 'robin', and so on. To store information with minimal redundancy, features common to a number of concepts are associated with the highest level they have in common, and lower-level nodes inherit all the features of higher-level nodes. For example, information such as 'can fly' and 'has feathers' would be associated with the 'bird' node, but could be accessed from lower nodes such as 'robin'. A parallel can be drawn between the concept of inheritance in semantic networks and in object-oriented programming languages such as Smalltalk (Goldberg and Robson, 1986), where changes made in parent classes of objects are propagated to subclasses. Inheritance and access to semantically related nodes in semantic networks occur through 'spreading activation' (Collins and Loftus, 1975), where activation of a source node spreads over time to allow retrieval of information embodied in other related nodes (for a discussion see Anderson, 1984). Semantic networks represent a long-term store of declarative knowledge in Anderson's (1983) ACT* model.

2.2.2 Schemas in semantic memory

At a higher level of organization, it has been proposed that semantic memory is organized into schemas (as discussed earlier). Alba and Hasher (1983) identify four features of schemas which dictate the contents of semantic memory. These are selection, abstraction, interpretation and integration. Activation of an existing schema selects new information for encoding, restricting it to salient or atypical events. For example, giving advance information to activate a relevant schema improves the recall of a complex abstract passage (Bransford and Johnson, 1972). Information in schemas is abstracted so that only the semantic content is stored and episodic details are lost, often leading to distortions in retrieval (Bransford *et al.*, 1972). Schemas use default values to make inferences and simplifications, and new information is integrated with the products of these interpretations to update or form new schemas. Alba and Hasher (1983) suggest that evidence for schema models of memory is equivocal. They argue that schemas cannot account for the richness of recall, and that semantic memory must contain an episodic component. However, schemas explain the flexibility of cognitive processing in the absence of complete information. For example, schematic organization accounts for the chunking of expert knowledge found in program recall studies (e.g. Adelson, 1981).

3 Programming as a problem-solving skill

Newell and Simon (1972) identify four features of a problem: the initial state; the goal state; the operators available for moving from the initial state to the goal; and restrictions on the operators. Programming may be seen as a problem-solving activity, where the initial state is the problem for which a program is required, and the goal state is both the solution which the program can calculate, and also the program itself. The operators consist of the syntactic and semantic features of the language and the cognitive skills of the programmer, and the operator restrictions are imposed by limitations of the language and the problem description. Problems are often described as having a 'state space', which consists of all the features of a problem, and all the possible moves that may be made between initial and goal states. This may be contrasted with a 'problem space' (Simon, 1978), which is the mental representation of a problem, together with the problem solver's prior knowledge. A goal of programming research is to examine differences between the state space and the problem space of the programming task.

3.1 Acquisition of cognitive skills

3.1.1 Expert-novice differences

One approach to studying cognitive skills is to compare the performance of novices and experts. Much of the research on expert-novice differences has examined programming, although other domains have been studied (e.g. physics, chess and medicine). Novices and experts differ in many aspects of problem solving, notably in the ways they represent problems, the strategies they choose for tackling problems, and the ways in which their knowledge about problem domains is organized. For example, differences in problem representation have been identified by Weiser and Shertz (1983). They found that novice programmers group programs according to superficial characteristics such as the application area, whereas experts group problems according to algorithms. Larkin (1981) found strategy differences, where novices solving physics problems tend to work backwards from the problem goal, whereas experts work forwards from the problem givens to the goal. Anderson (1985) suggests that in programming novices work forwards, writing a program line by line, whereas experts work backwards, breaking the program goal into modular units. Physics problems have a large number of givens, which are more predictive of the solution than are the problem goals. In programming, the problem givens are programming languages themselves, which are less predictive of the solution than the programming goal.

An expert-novice difference with important implications for programming research is the apparent schematic organization of expert knowledge (discussed further in Chapter 3.1). For example, studies of the recall of chess positions by novices and experts (Chase and Simon, 1973) showed that grandmasters recall game positions from long-term memory as 'chunks' of information. Grandmasters had greater recall of positions from realistic games than novices, but were no better than novices at recalling random positions. Similarly, McKeithen *et al.* (1981) found that experts recalled more lines from real Algol programs than novices, but there was no difference between experts and novices in the recall of scrambled programs. The difference between experts and novices increased over a number of trials, which contrasts with the findings of Chase and Simon (1973), who found that the advantage for experts declined. Grandmasters have chunks representing whole chess boards, but programs are too complex to be represented by individual chunks. Therefore it is necessary to reconstruct a program over a number of trials, by accessing schematic knowledge of programming concepts.

3.1.2 The ACT* model of skill acquisition

Another method of investigating cognitive skills is to study the changes an individual goes through in acquiring a skill. Anderson's (1983) ACT* model of skill acquisition is based on studies of individuals acquiring skills such as programming. Anderson proposes three stages in acquiring a cognitive skill. The first is a 'declarative' stage, where novices apply very general production rules to the declarative information given in a task or stored in long-term memory in order to solve a problem. The declarative stage does not commit the learner to task-specific procedures at too early a stage. However, the demands placed on the capacity of working memory by using general production rules requiring large amounts of declarative knowledge mean that performance is slow and error prone. A second stage of 'knowledge compilation' reduces the demands on the capacity of working memory by developing task-specific rules. This is achieved by 'composition', where production rules are combined into a single rule, and 'proceduralization', where task-specific information is added to production rules. The final stage is 'procedural' learning, where production rules are strengthened by a process of 'tuning', in which the speed and accuracy of rule application increases with practice.

Anderson *et al.* (1984) investigated the acquisition of Lisp programming skills by studying protocols (subjects' verbal reports and keystrokes made during problem solving) of novices learning to write functions. Figure 2 illustrates some examples of production rules which underlie different stages in the acquisition of Lisp programming skills. P1 is a rule applied by novices in problem solving by a process of analogy. A function is constructed by mapping the declarative knowledge of an example solution onto the declarative task information. Anderson (1987a) describes general production rules as 'weak method problem solutions'. These are used when a novice has no task-specific production rules, and must use very general methods to solve novel problems. Another example of a weak method problem solution is working backwards from a given solution. Knowledge compilation is illustrated by rules P2 to P5. Proceduralization of P2 produces a rule P3 which is specialized for operating on the head of a list. The composition of rules P3 and P4 produces a single rule P5 for adding the first element of one list to another list. P6 is an example of a 'tuned' rule that an expert would possess. Knowledge compilation may be characterized as 'success-driven' learning, that is, learning that results from the successful application of production rules. Anderson (1987b) argues that compilation often leads to one-trial learning, and that verbalization of performance is not possible once a procedural stage has been reached. Also, different production rules underlie different procedural tasks carried out in the same declarative domain. For example, McKendree and Anderson (1987) found that the skills acquired in learning to evaluate simple Lisp functions did not transfer to the task of writing Lisp functions.

3.1.3 The 'dynamic memory' approach to skill acquisition

The 'success-driven' learning in ACT* contrasts with 'failure-driven' learning in Schank's (1982) 'dynamic memory' theory, which is an extension of Schank and Abelson's (1977) scripts theory. Figure 3 shows a script of a possible program debugging session. Each event in the script is an instantiation of a 'scene', which is a generally defined sequence of actions. Scenes which commonly occur together are grouped into scripts by 'memory-organization packets' (MOPs) which represent high-

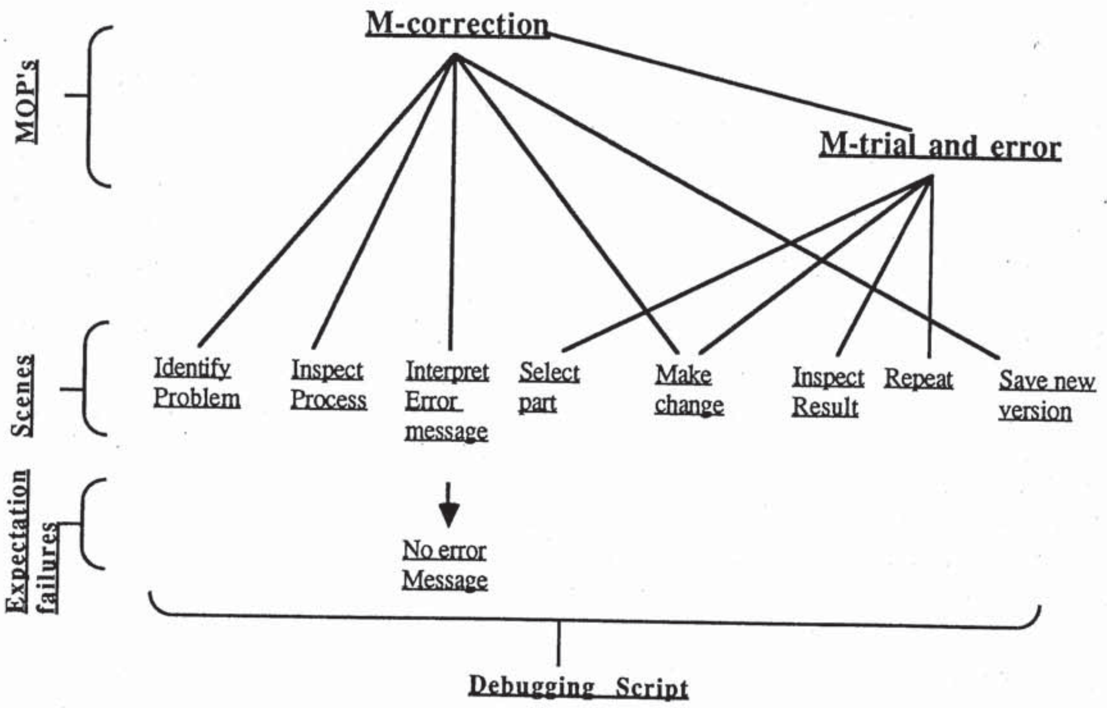

Figure 3: A hypothetical model of knowledge a novice might employ in a program debugging task, based on Schank's 'dynamic memory' theory (Schank 1982, p. 89). The MOPs (Memory Organization Packets) show two different sources of high-level knowledge that might generate expectancies in a debugging task. These MOPs may also guide performance in other tasks. For example, the MOP 'M-correction' might be accessed when mending a car, and the MOP 'M-trial and error' might be accessed when solving a maze puzzle. The five scenes generated by the MOP 'M-correction' represent events that generally occur together when correcting objects that function incorrectly. This MOP is linked to other MOPs which dictate strategies for making changes, such as the MOP 'M-trial and error'. The four scenes generated by the MOP 'M-trial and error' represent a possible novice debugging strategy, which is to generate potential corrections randomly and test them for success. The expectations raised by each scene in the script guide the debugging process. When an expectation fails, the MOP that organizes that scene must be adjusted to account for the failure. For example, the absence of an error message in running a faulty program may indicate a semantic rather than syntactic error. The novice must learn to discriminate between these sources of bugs, by adding additional scenes to the relevant MOPs.

level knowledge about scripted events. Figure 3 illustrates how MOPs which have different semantic contents (e.g. 'correction' and 'trial and error' MOPs) are used in the same task. MOPs control the learning of new skills by generating expectations about events. Learning is 'failure driven' in that it takes place when an expectation fails, under which circumstances the causal chain of events that led up to the failure is traced, and the MOP is then altered. ACT* and dynamic memory theory are not necessarily mutually exclusive, but may describe the learning of different skills. For example, learning to construct program code may be success driven, whereas debugging and testing may require failure-driven learning.

3.1.4 Transfer of problem-solving skills

Limitations of knowledge organization, representation and strategy provide major constraints on the problem space of a problem solver (for a review see Kahney, 1986). The acquisition of expertise overcomes these constraints and allows experts to tackle a broader range of related problems. One mechanism for doing this is to transfer skills from one problem domain to another. Transfer is an increasingly important area of research in problem solving, as well as in programming (discussed in Chapter 2.4). For example, Gick and Holyoak (1980) found that subjects were able to use an analogous example problem with a worked solution to solve Duncker's (1945) 'radiation problem'. Gick and Holyoak (1983) suggest that subjects learn to solve similar problems by using analogical transfer mechanisms, and develop schemas for particular classes of problem by 'schema induction'. Schemas are induced by forming an analogy between two or more training problems and solutions. A schematic representation of an abstracted solution is formed, which can then be evoked by the presence of appropriate context in a new problem.

Not all experiments have found successful transfer. Reed *et al.* (1985) found that transfer occurred only between equivalent algebra problems, and not between similar problems requiring a small manipulation of the problem solution. Positive transfer was found between similar problems, however, when a complex practice problem preceded a more simple transfer problem. This suggests that the transfer attributable to an analogy between equivalent practice and transfer problems is based on a shallow understanding of the problem features. When a deeper understanding of the problem is required, it is necessary to give practice problems where all the steps of the transfer problem solution are explicated. The conditions under which transfer may occur have important implications for training programming skills. For example, negative transfer of performance arising from inappropriate analogizing was found between Prolog comprehension tasks with different thematic content and representation (Ormerod *et al.*, 1990). The real-world familiarity of the training tasks prevented subjects from developing domain-independent comprehension strategies. Similarly, White (1988) found evidence of inappropriate transfer of knowledge by Pascal experts in solving Prolog debugging tasks. In other words, the Prolog task was carried out in terms of the subjects' Pascal knowledge.

3.2 Deductive reasoning

The process of constructing and testing a computer program may be compared to constructing and testing hypotheses by deductive reasoning. Deductive reasoning has traditionally been separated from other problem-solving activities, partly because formal 'competence' models exist for this skill. Piaget amongst others proposed that formal logic underlies human reasoning abilities (for a review see Gross, 1985), such that reasoning is carried by applying the rules of formal logic to propositions, independent of their real-world content or representation, to derive inferences. Thus, a programmer should be able to derive a program specification by formal logical reasoning, a process which should be unaffected by the programming domain or language notation. Indeed, some proponents of logic programming (e.g. Kowalski, 1982) have suggested that languages such as Prolog are closer to human-reasoning processes than conventional languages such as Pascal, because of an assumed match between the formal logic underlying Prolog and human reasoning.

A large body of empirical evidence exists showing systematic biases in reasoning, determined by the thematic content and representation of reasoning tasks (for a review see Evans, 1986). In the context of programming, this evidence is consistent with evidence of the influence of prior experience and problem representation on programming (e.g. Ormerod *et al.*, 1986; White, 1988). It also falsifies the arguments of Kowalski (1982) about the psychological advantages of logic programming languages. Biases in reasoning cannot be accounted for by a theory based on formal logic, but instead require a theorical explanation based on observed performance rather than logical competence. For example, Cheng and Holyoak (1985) found that a 'permission' rule such as 'if you are to drink alcohol, then you must be over eighteen' facilitated logical performance on Wason's (1966) selection task, where subjects must test the truth of the rule by selecting falsifying instances. They suggest that reasoning is carried out, not by logical inferences, but by 'pragmatic reasoning schemas' elicited by statements involving permission, obligation, causation and so on. These are generalizable knowledge structures containing rules for generating useful inferences.

3.2.1 'Mental models' theory

An influential theory which accounts for reasoning and language understanding without recourse to logic is Johnson-Laird's (1983) 'mental models'. In its most general form, a mental model is a mental representation of a problem space. It differs from state space and schematic representations in that, although individual units of information may be represented as propositions, a mental model is constructed out of propositions to form an analogue of the real world representation of a problem. In other words, a mental model has the same functional nature and structure as the system it models. For example, Mani and Johnson-Laird (1982) found that subjects recall the semantic content of spatial descriptions consistent with only one layout (e.g. 'the spoon is to the left of the knife; the plate is to the right of the knife', etc.), but recall verbatim details of spatial descriptions where a number of layouts are possible. This suggests that subjects construct a mental model of determinate descriptions, but resort to rote learning of unconnected propositions in indeterminate descriptions.

Johnson-Laird (1983) offers a 'procedural semantics' for the construction and searching of mental models in working memory, which allows errors in performance to be predicted on the basis of the order in which propositions are added to a mental model, and the number of alternative models which must be constructed. A general use of the term 'mental model', where it has the same relation structure as the real-world domain but a procedural semantics is not specified, is adopted by many theorists in problem solving (e.g. Gentner and Stevens, 1983), and in human-computer interaction (e.g. Manktelow and Jones, 1987). The creation and testing of mental models may explain aspects of expert programming skills, such as the interweaving of a number of programming plans into a single program (e.g. Rist, 1986).

3.3 Psycholinguistics and programming research

The review has for reasons of space been highly selective. The most obvious omission is a discussion of psycholinguistics (for a recent review, see Garnham, 1985). In part this is because many important concepts are covered in other topics, which illustrates how cognitive processes cannot be neatly partitioned as they were in early cognitive models. For example, schematic representations of knowledge have been used to account for text comprehension (Kintsch and van Dijk, 1978). Similarly, mental models theory has been used to account for the comprehension of sentences which require implicit inferences (Garnham, 1987). A competence versus performance debate has also occurred with language as well as reasoning (e.g. Chomsky, 1965). Theories of language understanding tend to emphasize either syntactic analyses (e.g. Fodor *et al.*, 1974) or semantic analyses (e.g. Schank, 1972). Programming research would seem amenable to a similar division of research areas. However, programming tends to be studied as a problem-solving rather than a linguistic activity, with notable exceptions (e.g. Sime *et al.*, 1977). Thus the focus of research into programming is not on syntactic features of different languages, but on semantic programming knowledge. However, as in recent theories of language understanding (e.g. Johnson-Laird, 1983), the interaction between semantic and syntactic components of the programming task has recently been highlighted (e.g. Arblaster, 1982).

4 Conclusions

The review of cognitive psychology presented in this chapter highlights the approaches to understanding human cognition which are of special relevance to programming research. Concepts that recur in many cognitive theories include schemas, production systems, limited resources, automation of skills with practice, working memory, semantic networks and mental models. Most employ propositional representations of one form or another, in which information is represented at a symbolic level.

A number of cognitive theorists (e.g. Fodor, 1983; Marr, 1982) argue that psychologists should concentrate their efforts on understanding processes such as perception, which appear to be carried out effortlessly and without error. They suggest that one should study the processes people are good at before tackling areas such as problem solving where people are slow and error prone. From an applied psychological perspective, it is just these areas where psychological research is at its most useful. Therefore, the motivations for psychological research into programming are strong, both for making programming an easier task, and for adding to a relatively unstable area of psychological theory.

References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition,* **9(4),** 422-433.

Alba, J.W., and Hasher, L. (1983). Is memory schematic? *Psychological Bulletin,* **93(2),** 203-231.

Anderson, J. R. (1983). *The Architecture of Cognition.* Cambridge, MA: Harvard University Press.

Anderson, J.R. (1984). Spreading activation. *In* J. R. Anderson and S. M. Kosslyn (Eds.), *Tutorials in Learning and Memory.* New York: Freeman.

Anderson, J. R. (1985). *Cognitive Psychology and its Implications,* 2nd edn. New York: Freeman.

Anderson, J. R. (1987a). Methodologies for studying human knowledge. *Behavioural and Brain Sciences,* **10(3),** 467-505.

Anderson, J. R. (1987b). Skill acquisition: compilation of weak-method problem solutions. *Psychological Review,* **94(2),** 192-210.

Anderson, J. R. and Jeffries, R. (1985). Novice LISP errors: undetected losses of information from working memory. *Human-Computer Interaction,* **1,** 107-131.

Anderson, J.R., Farrell, R., and Sauers, R. (1984). Learning to program in Lisp. *Cognitive Science,* **8,** 87-129.

Anzai, Y. and Simon, H. A. (1979). The theory of learning by doing. *Psychological Review,* **86(2),** 124-140.

Arblaster, A.T. (1982). Human factors in the design and use of computing languages. *International Journal of Man-Machine Studies,* **17,** 211-224.

Atkinson, R. C. and Shiffrin, R. M. (1968). Human memory: a proposed system and its control processes. *The Psychology of Learning and Motivation,* vol 2. New York: Academic Press.

Baddeley, A. and Hitch, H. (1974). Working memory. *In* G.H. Bower (Ed.), *The Psychology of Learning and Motivation,* vol 8. New York: Academic Press.

Bartlett, F. C. (1932). *Remembering: A Study in Experimental and Social Psychology.* London: Cambridge University Press.

Bransford, J. D. and Johnson, M. K. (1972). Contextual prerequisites for understanding: some investigations of comprehension and recall. *Journal of Verbal Learning and Verbal Behavior,* **11,** 717-726.

Bransford, J. D., Barclay, J. R. and Franks, J. J. (1972). Sentence memory: a constructive versus interpretive approach. *Cognitive Psychology,* **3,** 193-209.

Broadbent, D. E. (1958). *Perception and Communication.* Oxford: Pergamon Press.

Chase, W. G. and Simon, H. A. (1973). The minds eye in chess. *In* W. G. Chase (Ed.), *Visual Information Processing.* New York: Academic Press.

Cheng, P. W. and Holyoak, K. J. (1985). Pragmatic reasoning schemas. *Cognitive Psychology,* **17**, 391-416.

Chomsky, N. (1965). *Aspects of the Theory of Syntax.* Cambridge, MA: MIT Press.

Cohen, G., Eysenk, M. W. and Le Voi, M. E. (1986). *Memory: A Cognitive Approach.* Milton Keynes: Open University Press.

Collins, A. M. and Loftus, E. F. (1975). A spreading-activation theory of semantic processing. *Psychological Review,* **82**, 407-428.

Collins, A. M. and Quillian, M. R. (1970). Does category size affect categorization time? *Journal of Verbal Learning and Verbal Behavior,* **9**, 432-438.

Duncker, K. (1945). On problem solving. *Psychological Monographs,* **58(270)**, 1-113.

Evans, J. St. B. (1986). Reasoning. *In* H. Beloff and A. M. Colman (Eds), *Psychology Survey,* vol 6. Leicester: British Psychological Society.

Eysenk, M. W. (1984). *A Handbook of Cognitive Psychology.* London: Lawrence Erlbaum Associates.

Fodor, J. A. (1983). *The Modularity of Mind.* Cambridge MA: Bradford Books/MIT Press.

Fodor, J. A., Bever, T. G. and Garrett, M. F. (1974). *The Psychology of Language.* New York: McGraw-Hill.

Galotti, K. M. and Ganong, W. F. (1985). What non-programmers know about programming: natural language procedure specification. *International Journal of Man-Machine Studies,* **22**, 1-10.

Garnham, A. (1985). *Psycholinguistics: Central Topics.* London: Methuen.

Garnham, A. (1987). *Mental Models as Representations of Discourse and Text.* Chichester: Ellis Horwood.

Gentner, D. and Stevens, A. L. (1983). *Mental Models.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Gick, M. L. and Holyoak, K. J. (1980). Analogical problem solving. *Cognitive Psychology,* **12**, 306-355.

Gick, M. L. and Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive Psychology,* **15**, 1-38.

Goldberg, A. and Robson, D. (1986). *Smalltalk 80 – The Language and its Implementation.* Addison-Wesley.

Gross, T. F. (1985). *Cognitive Development.* California: Wadsworth.

Johnson-Laird, P. N. (1983). *Mental models.* London: Cambridge University Press.

Kahney, H. (1986). *Problem Solving: A Cognitive Approach.* Milton Keynes: Open University Press.

Kintsch, W. and van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review,* **85**, 363-394.

Kowalski, R. (1982). Logic programming for the fifth generation. *In* K. L. Clark and S.-A. Tärnlund (Eds), *Logic Programming.* Amsterdam: North Holland.

Larkin, J. (1981). Enriching formal knowledge: A model for learning to solve textbook physics problems. *In* J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

McKeithen, K. B., Reitman, J. S., Rueter, H. H. and Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Canadian Journal of Psychology,* **13**, 307-325.

McKendree, J. and Anderson, J. R. (1987). Effect of practice on knowledge and use of basic Lisp. *In* J. M. Carroll (Ed.), *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.* Cambridge, MA: MIT Press.

Mani, K. and Johnson-Laird, P. N. (1982). The mental representation of spatial descriptions. *Memory and Cognition,* **10**, 477-488.

Manktelow, K. I. and Jones, J. (1987). Principles from the psychology of thinking and mental models. *In* M. M. Gardiner and B. Christie (Eds), *Applying Psychology to User-Interface Design.* Chichester: Wiley.

Marr, D. (1982). *Vision: A Computational Investigation in the Human Representation of Visual Information.* San Fransisco: Freeman.

Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for information processing. *Psychological Review,* **63**, 81-97.

Miller, L. A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies,* **6**, 237-260.

Minsky, M. (1975). A framework for representing knowledge. *In* P. H. Winston (Ed.), *The Psychology of Computer Vision.* New York: McGraw-Hill.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving.* Englewood Cliffs: Prentice-Hall.

Ormerod, T. C., Manktelow, K. I., Robson, E. H., and Steward, A. P. (1986). Content and representation effects with reasoning tasks in Prolog form. *Behaviour and Information Technology,* **5(2)**, 157-168.

Ormerod, T. C., Manktelow, K. I., Steward, A. P. and Robson, E. H. (1990). The effects of content and representation on the transfer of Prolog reasoning skills. *In* K.J. Gilhooly, M.T.G. Keane, R.H. Logie and G. Erdos (Eds), *Lines of Thinking.* Chichester: Wiley.

Paivio, A. (1971). *Imagery and Verbal Processes.* New York: Holt, Rinehart and Winston.

Pylyshyn, Z. W. (1979). Imagery theory: Not mysterious – just wrong. *The Behavioral and Brain Sciences,* **2**, 561-563.

Pylyshyn, Z. W. (1984). *Computation and Cognition.* Cambridge, MA: Bradford Books/MIT Press.

Reason, J. T. (1979). Actions not as planned: the price of automatization. *In* G. Underwood and R. Stevens (Eds), *Aspects of Consciousness.* London: Academic Press.

Reed, S. K., Dempster, A. and Ettinger, M. (1985). Usefulness of analogous solutions for solving algebra word problems. *Journal of Experimental Psychology: Learning, Memory, and Cognition,* **11(1)**, 106-125.

Rist, R. S. (1986). Plans in programming: definition, demonstration, and development. *In* E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Rumelhart, D. E. and McClelland, J. L. (Eds). (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* vol. 1: Foundations. Cambridge, MA: Bradford Books/MIT Press.

Rumelhart, D. E. and Norman, D. A. (1983). Representation of knowledge. Technical Report No. 116, Center for Human Information Processing, San Diego, University of California. Reprinted in Aitkenhead, A. M. and Slack, J. M. (Eds) (1985). *Issues in Cognitive Modelling.* London: Lawrence Erlbaum Associates.

Schank, R. C. (1972). Conceptual dependency: a theory of natural language understanding. *Cognitive Psychology,* **3**, 552-631.

Schank, R. C. (1982). *Dynamic Memory.* New York: Cambridge University Press.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Searle, J. (1980). Minds, brains, and programs. *The Behavioral and Brain Sciences,* **3**, 417-457.

Shiffrin, R. and Schneider, W. (1977). Controlled and automatic human information processing: II. Perceptual learning, automatic attending, and a general theory. *Psychological Review,* **84**, 127-190.

Sime, M. E., Arblaster, A. T. and Green, T. R. G. (1977). Reducing errors in programming conditionals by prescribing a writing procedure. *International Journal of Man-machine Studies,* **9**, 119-126.

Simon, H. A. (1978). Information processing theories of human problem solving. *In* W. K. Estes (Ed.), *Handbook of Learning and Cognitive Processes.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Soloway, E. and Erlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering,* **10(5)**, 595-609.

Tulving, E. (1985). How many memory systems are there? *American Psychologist,* **40**, 385-398.

Wason, P. C. (1966). Reasoning. *In* B. M. Foss (Ed.), *New Horizons in Psychology,* vol. 1, Harmondsworth: Penguin.

Weiser, M. and Shertz, J. (1983). Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies,* **19**, 391-398.

White, R. (1988). Effects of Pascal knowledge on novice PROLOG programmers. DAI research paper No. 399, Department of Artificial Intelligence, University of Edinburgh.

Wilensky, R. (1981). Meta-planning: Representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science,* **5**, 197-233.