Tools for Creating Tools: Programming in Artificial Intelligence

PHILIP MACHANICK
Institute for Maritime Technology
P.O. Box 181
SIMON'S TOWN
7995

SUMMARY

After a brief look at how the notion of the stored program is applied by programmers in the area of Artificial Intelligence (AI), we shall look at a specific AI tool: a production system. The discussion is at a non-rigorous level, intended to allow the concepts involved to be related to other fields more familiar to those in the mainstream of Computer Science, most notably, the field of Compiler Construction. Production systems are of particular interest because they are the basis on which rule-based expert systems are constructed. To complete the picture, future developments in tools for building expert systems are considered.

THE STORED PROGRAM

In the "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" of 1946, Burks, Goldstine and Von Neumann regarded the need for address modification as one of the most important reasons for storing instructions in the same kind of memory as that used for data. The conception of index registers by Kilburn in 1951 rendered this concern obsolete [Goldstine 1972].

The stored program concept is still with us today for a completely different reason (though Von Neumann was heading in the right direction with his notion of self-reproducing automata [*ibid.*]): it enables us to create programs which function as tools for creating other programs. Some examples are assemblers, compilers, operating systems, linkers, editors and symbolic debuggers. And yet, outside AI (and a few specialized areas, such as Compiler Construction), the idea that a program is like data, in the sense that it can be built up by a program, has received scant recognition.

PROGRAMS AS DATA IN LISP

LISP programs are not particularly readable. A function definition (in an early implementation - later versions have improved considerably) could look like this:

```
(DEFINE (QUOTE
  (
    (double (LAMBDA (anumber)
       (PLUS anumber anumber)
    ))
  )
))
```

This function doubles its argument, e.g., the call

```
(double 10)
```

would return

```
20
```

Even with careful use of layout to simplify counting parentheses, the definition is not particularly readable. Newer, more readable implementations [Winston and Horn 1981] reduce the problem, but programming directly in LISP remains an unpleasant task.

The strength of the language lies in the similarity between the representation of programs and data structures, and the ready access the programmer has to the interpreter through the functions EVAL and APPLY. For example, it would not be very difficult (using standard list-manipulation functions) to convert a more readable-looking function definition into the definition given above. This "standard" definition could then be evaluated using EVAL.

With function definitions looking like this:

```
(prettydefine
    double (anumber)
   (PLUS anumber anumber)
)
```

the programmer's task would be less of an ordeal.

A facility for executing data, i.e., passing it to the interpreter, is also available in APL and SNOBOL, but has not achieved the same prominence among programmers using these languages as it has in LISP [Machanick 1980b].

Most AI researches regard LISP as a kind of "assembly language" on top of which an application language must be created to solve the problem in hand [Machanick 1980a]. While some languages (e.g., KRL [*ibid.*], OPS-5 [Duda and Gaschnig 1981]) have achieved recognition among researchers other than their creators, many languages only exist as part of the implementation of a specific application. This explains why LISP (in one form or another) has remained the main AI language for so long - AI workers do not program in LISP, they use LISP as the underlying virtual machine on which to implement a higher level application-specialised language. Since AI is largely experimental (despite the advent of commercially viable natural language systems [Eisenberg and Hill 1984, Johnson 1984], games and expert systems), it is not surprising that there is a considerable variation in control structures and data structures from one application to another. The resultant need for flexibility in AI languages has contributed to the practise of building specialised tools on top of the LISP virtual machine.

PRODUCTION SYSTEMS: AN INTRODUCTION

Production systems are an important AI tool. They form the basis for constructing rule-based expert systems, and are representative of an important class of programs: those which are data driven as opposed to procedural. This broad class includes object-oriented languages (such as Smalltalk [Robson and Goldberg (editors) 1981]) and the data flow model [Sharp 1980].

The essential components of a production system are

- rules (knowledge base)
  in the simplest case, an ordered pair of strings of the form LHS → RHS (also written as if LHS then RHS, or RHS if LHS)

- data base
  the current state of the system: data describing the problem in hand (possibly modified by execution of the program) as opposed to rules, which describe how to solve such a problem

interpreter
: depending on the strategy used, either attempts to match the LHS or the RHS of rules to the data base (respectively, forward or backward chaining); the data base is updated according to information provided by the rule selected

[Davis and King 1977]

In practice, things can become very complex. Exactly what happens when a "match" is attempted could be anything from a simple string comparison to a sophisticated pattern match (e.g., matching the data base's patient symptoms etc. against rules for diagnosing an infectious disease). Furthermore, the action on the data base on matching could vary from simply replacing or deleting some part of the data base to the result of a complex piece of computation.

One of the essential features of a production system is that each rule is meant to represent a separate piece of knowledge [*ibid.*] . To the extent that all knowledge is interrelated, this may be an unattainable ideal. Nonetheless, production systems can be made to function in such a way to be relatively robust when rules are altered, added or deleted. An essential requirement is that the knowledge being represented be possible to see as consisting of a wide range of separate pieces of knowledge, as opposed to a unified theory. In the latter case, a more procedural approach may be appropriate: a production system attempting to solve such a problem would either end up with complex function calls being required when a match was found (which is regarded as being contrary to the spirit in which a production system should be designed) or with cumbersome rules [*ibid.*] .

A production system executes by cycling through a process of testing rules against the data base for matches, until a goal state is reached, or no more matches can be made. If more than one rule matches, conflict resolution - which varies from one implementation to another - is applied to select a single rule. This differs from a procedural approach, in that the order in which pieces of the program is written does not affect the order of execution. In a "pure" production system, the order of execution depends only on the current state of the data base, which can lead to unpredictable behaviour. Some determinancy can be added, reducing the independence of rules from each other, by having rules add control elements to the data base. This is another approach which should not be necessary if a production system is used in the way in which it was originally intended to be used [*ibid.*] .

Without going more deeply into the subject, we shall work through an example to illustrate the essential principles - especially backward chaining (also known as consequent reasoning, or goal-driven programming) and forward chaining (antecedent reasoning, or data-driven programming).

EXAMPLE

problem check if a string of [],(),{} is correctly nested

notation for rules

general : LHS → RHS

goal state S: S → .goal

termination conditions

matching .goal, or .error when no other match is possible

variables

?X : a string of length zero or more - ?X on the left-hand-side and on the right-hand-side of the same production (i.e., rule) represents the same string

conflict resolution

simple approach: try rules in the order written, and stop as soon as a match is found

backtracking

backward chaining: backtrack if the goal string becomes longer than the data base

forward chaining: backtrack if the data base does not match any of the rules

definitions of control strategies [Duda and Gaschnig 1981]

general form of a rule:

$$\text{antecedant}_1 \ldots \text{antecedant}_m \rightarrow \text{consequent}_1 \ldots \text{consequent}_n$$

forward chaining

scan through the rules for one with antecendents (LHS) matching the data base, apply the rule, update the data base (here, the action taken is to replace the entire data base by the consequents - RHS - of the rule), repeating until a goal state is reached or there are no more applicable rules - indicated here by .error in the data base

backward chaining

select a goal: scan rules for those whose consequents (RHS) could achieve the goal; if the antecedants match the data base, a solution has been found; if an unmatched antecedant is found, matching it recursively becomes the new subgoal; if there are no rules for a subgoal, ask the user for more facts for the data base

rules	comments
(1) DONE → .goal	the only goal
(2) [] → DONE (3) () → DONE (4) {} → DONE	handle the trivial cases directly
(5) [?X] → ?X (6) (?X) → ?X (7) {?X} → ?X	reduce non-trivial cases to the trivial case recursively
(8) ?X → .error	only reach this rule if no others match

An execution trace is given as Figure 1.

execution trace

initial data base: { ([ ]) }

forward chaining			backward chaining	
rule	antecendent	new data base	goal	matching consequents
(7)	{?X}	([])		
(6)	(?X)	[]	DONE	(2),(3),(4) match - try (2):
(2)[1]	[]	DONE	[]	(5),(6),(7) match - try (5):
(5)[1]	[?X]	<empty>	[[]]	(5),(6),(7) match - try (5):
			[[[]]]	
			:	
			goal longer than data base: backtrack	
			[[]]	try (6):
			([[]])	
			:	
			backtrack	try (7):
			{[[]]}	
			:	
			backtrack	
			:	
			eventually back to	
			[]	try (6):
			([])	(5), (6), (7) try (5):
			[([])]	
			:	try (6):
			(([]))	
			:	try (7):
			{([])}	*match*

[1]conflict resolution: choose (2)- DONE is a goal so STOP

NOTE: if we had made the wrong choice, we would have to backtrack when the <empty> data base matched rule (8) and became .error, but this would have presented the difficulty of restoring the data base to the state it was in before the "incorrect" rule was selected; in this case, backtracking could also have been eliminated by adding the rule <empty> → DONE (which represents the "piece of knowledge" that the empty string contains no unmatched parenthesis)

Figure 1: Execution Trace

PRACTICAL CONSIDERATIONS

Although the example makes forward chaining look simpler and more efficient than backward chaining (which is supported by experience in the field of parsing formal languages, where bottom-up parsing is more efficient and more general than top-down parsing [Aho and Ullman 1977]), aspects of a specific implementation may tilt the balance the other way. It has been asserted that backward chaining corresponds more closely to how humans solve problems [Davis and King 1977]; the same could be said of top-down parsing.

Confusingly, both the goal-oriented (backward chaining) [*ibid.*] and forward chaining [Buchanan 1982] approaches have been claimed to be preferable from the point of view of predicting the effects of altering the knowledge base (i.e.,the rules).

Clearly, there is much more work still to be done in this area. In any case, such considerations only effect the efficiency of the finished product; the

biggest bottleneck remains transferring knowledge from the source (the "expert") to the knowledge base (the rules) [Buchanan 1982].

WHAT OF THE FUTURE?

PROLOG [Clark and McCabe 1982] has two major drawbacks: it only implements backward chaining (which may turn out not to be the method of choice), and the order of rules is significant. The former factor can readily lead to programs with infinite loops, which have to be eliminated by heuristics (such as that applied to decide when to backtrack in the worked example) or by rewriting the rules (in a way analogous to eliminating left recursion in a top-down parser [Aho and Ullman 1977]). The need to consider the order of the rules further detracts from the appealing conceptual simplicity of the PROLOG approach.

Still, PROLOG is the implementation language of the Japanese Fifth Generation project [Feigenbaum and McCorduck 1983] and these problems are receiving due attention. In the UK, for example, ways of introducing concurrency, non-classical logics and a more friendly user interface are being researched [Durham 1984]. In summary, PROLOG is a higher level tool than LISP: at the expense of flexibility, it relieves the programmer of the need to construct a production system interpreter.

More general or powerful tools (such as KAS, AGE and HEARSAY III [Duda and Gaschnig 1981]) are less widely available, present the uninitiated with a bewildering array of alternatives and are unlikely to fit on the current generation of micros. Ultimately, the question is whether an experimental or production expert system is being constructed: with PROLOG in its present form, the system may be easy to get off the ground (good for experimental purposes) but could run into problems because the language designers have pre-empted too many decisions (bad for making profits).

Although PROLOG is not without fault, as a means of entry to the area of expert systems, it has much to recommend it. If future developments rectify its known faults, it could become widely used. But, like so many languages which have grown on an experimental basis, the likelihood is that there will be a proliferation of incompatible dialects.

REFERENCES

Aho, Alfred V. and Ullman, Jeffrey D., *Principles of Compiler Design*, Addison-Wesley, Reading, Masschusetts, 1977.

Buchanan, B.G., New Research on Expert Systems, *Machine Intelligence* **10** (269-299),Ellis Horwood, Chichester, 1982. Editors: Hayes, J.E., Michie, Donald and Pao, Y-H. Also published as: *Research on Expert Systems*, Report No. STAN-CS-81-837, Department of Computer Science, Stanford University, Stanford, California, March 1981.

Clarke, K.L. and McCabe, F.G., PROLOG: A Language for Implementing Expert Systems, *ibid.* (455-475).

Davis, Randall and King, Jonathan., An Overview of Production Systems, *Machine Intelligence* **8** (300-332). Ellis Horwood, Chichester, 1977. Editors: Elcock, E.W. and Michie, Donald.

Duda, Richard and Gaschnig, John G., Knowledge-Based Expert Systems Come of Age, *Byte* **6**(9) September 1981 (238-281).

Durham, Tony., Logical Approach to PROLOG, *ComputerWeek* **7** (18) 7 May 1984 (15-15).

Eisenberg, Jane and Hill, Jeffrey., Using Natural-Language Systems on Personal Computers, *Byte* **9**(1) January 1984 (226-238).

Feigenbaum, Edward A. and McCorduck, Pamela., *The Fifth Generation: Artificial Intelligence and Japan's Computer Challenge to the World*, Addison-Wesley, Reading, Massachusetts, 1983.

Goldstine, Herman H., *The Computer from Pascal to Von Neumann*, Princeton University Press, Princeton, New Jersey, 1972.

Johnson, Jan., Easy Does It, *Datamation* **30**(9) 15 June 1984 (48-60).

Machanick, Philip.
a. Communicating with Thinking Machines: A Review of Progress in Artificial Intelligence Languages, Computer Science Honours Essay, University of the Witwatersrand, Johannesburg, 1980. 17 pages.

b. Programming Languages: Progress and Future, Computer Science Honours Project, University of the Witwatersrand, Johannesburg, 1980.

Robson, Dave and Goldberg, Adele. (editors) The Smalltalk-80 System, *SIGPLAN Notices* **15**(9) September 1980 (44-57).

Winston, Patrick Henry and Horn, Berthold Klaus Paul. 1981, *LISP*, Addison-Wesley, Reading, Massachusetts, 1981.