7

System Architectures for Prolog Execution*

Markian M. Gooley
Benjamin W. Wah

INTRODUCTION

Motivation

Expert systems are arguably the most successful and lucrative result of research in artificial intelligence. However, they generally are executed on conventional computers, not on machines designed to support the operations they frequently must perform. Worse yet, languages for writing expert systems, or *shells,* often have limited power [57] and are interpreted—i.e., executed by a program running on a computer—rather than compiled to the computer's actual instruction codes. As a result, expert systems tend to execute slowly and be limited by their shells [24]. Writing them in a general-purpose language improves their speed and versatility, but makes them harder to design, implement, and modify.

Why should those who study or implement expert systems take interest in Prolog? Prolog is no panacea, and it does not have some features built into it that a shell might have, such as the ability to handle uncertain information or generate explanations [24]. However,

1. Prolog has its own inference (backward-chaining), and search (depth-first) mechanisms [45], like a shell.
2. Unlike many shells, it also has the computational power of first-order logic [26].

* This work was supported in part by the National Aeronautics and Space Administration under grants NAG 1-613 and NCC 2-481.

3. Shells can readily be written in Prolog, though with some loss of speed. Such shells can use a different type of inference, handle uncertain information, provide explanations, and so forth [45].
4. A machine architecture (the *Warren abstract machine*) [54] already exists for Prolog and can be emulated in software or microcode or built as hardware; it is also the building block for Prolog multiprocessors [56]. Anyone designing a machine for expert systems will profit from familiarity with machines designed for Prolog.

Various authors have argued that Prolog should be used more widely for expert systems [5, 19, 45]. They cite some of the aforesaid arguments and suggest that expert systems written in Prolog require less code and execute more quickly than systems written in other languages. Most such Prolog expert systems, however, are still written where Prolog is popular (e.g., Japan), or in association with members of the logic programming community.

Prolog has been used to build expert systems for all the usual applications. An especially popular one is the design of circuits or computers: VLSI layouts [28], compacting microcode [39], and so forth. Other applications include diagnostic and therapeutic medicine [29], credit evaluation [3], and strategy for games, law [36, 59], and negotiation [30].

Overview

The intent of this chapter is to acquaint the reader with Prolog and Prolog machines, especially Warren's. First we introduce the language, beginning with simple examples and proceeding to more exact definitions. Then, again using examples, we describe Prolog's method of execution and some features that help suit it to expert systems. Next we describe the parts of the Warren abstract machine (WAM) [54], and show how it executes programs. We then consider parallel execution. After reviewing some of the parallel architectures proposed for expert system machines, we return to Prolog to look at the principal varieties of parallelism available and examples of the architectures which exploit them. Finally, we summarize our presentation and give our thoughts on machines for expert systems.

THE PROLOG LANGUAGE

As a supplement to this discussion, the reader may wish to consult an introductory text such as Clocksin and Mellish [7] or Sterling and Shapiro [45].

Tutorial Introduction

Facts. A *fact* states a relationship between objects. An expert system implemented in Prolog has a knowledge base of facts. For example, here is one way to represent a family tree:

```
Husband = charles
Wife = jane ;

Husband = george
Wife = lydia ;

Husband = fitzwilliam
Wife = elizabeth ;

no
```

Prolog looks at the facts for `wife` in the order that they are written, successively binding the variables `Husband` and `Wife` to husband-wife pairs. A similar query with only one variable gives the expected result:

```
?- wife(Husband,elizabeth).

Husband = fitzwilliam ;

no
```

Conjunctions. Prolog allows conjunctive queries, which have answers if all of their components do. The conjunction is executed from left to right. For example, suppose that we want to know whether `mrs_bennet` has any married daughters. Then the following is the query (note that the "," between its parts is a conjunction and is *not* the same as the commas within the tuples):

```
?- mother(D,mrs_bennet), wife(_,D).

D = jane ;

D = lydia ;

D = elizabeth ;

no
```

Prolog scans the facts for `mother` in order, binding `D` to `jane`. Then it scans the facts for `wife`, testing whether `jane` is a wife; she is. (Because we do not care who the husband is, we use "_", the *anonymous variable*, which matches anything and is never bound). When we ask for another answer with ";", Prolog seeks another husband-wife pair (it knows nothing of laws forbidding polygamy), fails to find one, and *backtracks* to `mother` to find another child of `mrs_bennet`. There it finds `lydia`, and so it again scans for facts for `wife`. `lydia` is also a wife. Again we force backtracking, and again Prolog scans the rest of `wife`, fails, backtracks to `mother` to find `elizabeth`, and proceeds to `wife` to find that she is a wife. We use ";", `wife` yields failure, so does `mother`, and Prolog returns `no`.

Rules. Prolog lets us define *rules* by which it can infer new relations from facts. We can base a rule on the query of the previous example:

```
married_daughter(Mother,Daughter):-
        mother(Daughter,Mother),
        wife(_,Daughter).
```

We can now use this rule instead of the compound query:

```
?- married_daughter(mrs_bennet,D).
```

Some relations require multiple rules, which give a *disjunction*. For instance, a parent is either a mother or married to a mother (assuming no illegitimacy):

```
parent(Child,Parent):- mother(Child,Parent).
parent(Child,Parent):-
        wife(Parent,Mother),
        mother(Child,Mother).
```

(Note that a variable is local to its rule: `Child` in the first rule has no relation to `Child` in the second.) Now consider the following query:

```
?- parent(jane,Parent).

Parent = mrs_bennet ;

Parent = mr_bennet ;

no
```

Prolog gets an answer from the first rule; then, when we use ";", it tries to get another answer, fails, and then uses the second rule, which yields another answer to ";" before failing.

Definitions

The foregoing should provide a "feel" for the rudiments of Prolog: information is stored as facts and retrieved using queries; variables and conjunction make useful queries possible; and rules define ways to infer further information. The examples also show something of Prolog's mechanism of execution.

Terms. We now proceed to define things more rigorously. The *term* is the basic unit of Prolog's syntax. A term is a *constant*, a *variable*, or a *compound term*. Constants are *numbers* or *atoms*, where an atom is an alphanumeric (including the character "_") string starting with a lowercase letter. Variables are alphanumeric strings starting with an uppercase letter or "_", with the solitary "_" reserved for the anonymous variable.

A compound term is an atom, called a *functor*, together with some number of *arguments* (its *arity*), which are also terms; e.g., the term `f(g,h(I,J),k)` has functor `f`, an arity of 3, and arguments `g`, `h(I,J)`, and `k`. Sometimes we consider an atom a compound term of arity 0. *Ground terms* contain no variables.

Clauses. The facts, queries, and rules we have just seen are special cases of the Prolog *clause*. A clause has the form

```
head:- goal_1, goal_2, . . . , goal_n.
```

where the `head` and the `goals` are terms. The "`:-`" is a stylized arrow representing implication, because the clause is in fact a *Horn clause*, i.e., a conjunction of premises (goals) implying a single conclusion (head). The *logical* meaning of the clause is "The `head` holds *if* `goal_1` *and* `goal_2` *and* all the goals through `goal_n` hold." We can also consider the clause to have a *procedural* meaning, viz., "To do the task indicated by `head`, do `goal_1` *and* `goal_2` *and* all the goals through `goal_n`." [45] The goals form the *body* of the clause.

A fact, then, is a clause whose body is always true (hence omitted); e.g., `girl(mary).` is actually `girl(mary):-true.`, where `true` is a goal that always succeeds. Thus, `girl(mary).` means "`girl(mary)` holds if `true` holds," or "To prove `girl(mary)`, prove `true`." A query lacks a head. It has the form

```
?- goal_1, goal_2, . . . , goal_n.
```

but the "`?-`" is the same as a "`:-`"; i.e., we actually have

```
false:- goal_1, goal_2, . . . , goal_n.
```

In giving Prolog a query, we issue a challenge: "I say that this conjunction of goals implies falsity. Prove constructively that I'm wrong." A rule is simply the most general form of clause.

Predicates and programs. A group of clauses whose heads have the same functor and arity is considered a *predicate*. A predicate is the disjunction of its clauses, as we saw in the example for `parent`; it is the Prolog version of a procedure. It can consist of facts, rules, or both. Often we refer to a predicate by its name and arity, i.e., *name/arity*, as in `parent/2`. A collection of predicates forms a *program*.

Prolog's Mechanism of Execution

We have already seen a couple of simple examples of Prolog execution. We now describe the general mechanism and give examples that are more complex.

Unification. Suppose that we begin with a query with a single goal:

```
?- q(a(X),Y).
```

Prolog tries to match this with the heads of successive clauses of the predicate q/2:

```
q(b,X):- r(X), s(X).
q(a(X),b):- s(X).
q(a(b(c,d)),e).
```

Only if its head *unifies* with the calling goal will the rest of a clause be executed. Unification is an attempt to make two terms identical by instantiating variables; the set of instantiations is called the *most general unifier*. Thus, `q(a(X),Y)` does not unify with `q(b,X)`, because no instantiation can make their first arguments the same (note again that the `X` in the first term and the `X` in the second term are distinct). However, `q(a(X),Y)` unifies with `q(a(X),b)`: the distinct `X`s are made equivalent, `Y` is set to `b`, and the second clause is executed. If backtracking reaches the third clause, goal and fact unify, with `X = b(c,d)` and `Y = e`.

Forward execution. Forward execution of Prolog begins with a query. We can think of Prolog as maintaining a list of goals to execute [45]. It takes the leftmost goal of this list and tries to unify it with the head of a clause. If the unification succeeds, Prolog replaces the goal with the goals forming the body of the clause matched, keeping track of the instantiations involved. Then it repeats the process for the new leftmost goal. For instance,

```
?- q1, q2, q3.
```

becomes

```
r1, r2, r3, r4, q2, q3.
```

and then

```
s1, s2, r2, r3, r4, q2, q3.,
```

given appropriate clauses. (For brevity, we show no variables.) Eventually the leftmost goal matches a fact. Because facts have no bodies (strictly speaking, they have bodies with the trivial goal `true`), Prolog can delete the current goal and proceed to the next one. Hence, suppose that `s1` and `s2` match facts; then the list of goals becomes shorter:

```
<s1 matches...>
s2, r2, r3, r4, q2, q3.
<s2 matches...>
r2, r3, r4, q2, q3.
```

When the list of goals becomes empty, the original query is solved and Prolog returns the instantiations of variables appearing in it.

Backward execution. If some goal cannot match any clause head, or if the user asks for another solution to a query, Prolog must backtrack. Returning to the preceding example, suppose that no clause matches `r2`; then Prolog backtracks to `s2` thus:

```
r2, r3, r4, q2, q3.
<backtracking...>
s2, r2, r3, r4, q2, q3.
```

Prolog undoes any instantiations done by `s2` and then tries to unify `s2` with the next untried clause of the predicate it calls. If it succeeds for some remaining clause, we might have the following:

```
t1, t2, t3, r2, r3, r4, q2, q3.
```

If not, Prolog backtracks to `s1`, undoing any instantiations done by `s1`, and tries the next clause of the predicate it calls. If it then succeeds, we might have the following:

```
s1, s2, r2, r3, r4, q2, q3.

u1, u2, s2, r2, r3, r4, q2, q3.
```

If not, we must backtrack to `r1`:

```
r1, r2, r3, r4, q2, q3.
```

If we end up backtracking to the original query and no alternatives remain to match its leftmost goal, the query fails.

Features of Prolog

Data structures and recursion. Data structures are constructed from compound terms. We can represent a binary tree, for example, using a term t(< *Left* >, < *Right* >), where < *Left* > and < *Right* > are either leaves or subterms representing subtrees. Thus, the tree shown in Figure 7–1 is represented in Prolog by

```
t(1, t(t(2, 3), t(t(4, t(5, 6)), 7)))
```

Prolog has a special functor, ".", for constructing *lists*, and a special atom, "[]", to represent the empty list. The list of the first five natural numbers is therefore `.(1,.(2,.(3,.(4,.(5,[])))))`, but for convenience it is written `1,2,3,4,5`.

Handling such recursive data structures as trees and lists requires recursive predicates. This ubiquitous one appends two lists to create a third:

```
append([],L,L).
append([H|T],L,[H|T2]):- append(T,L,T2).
```

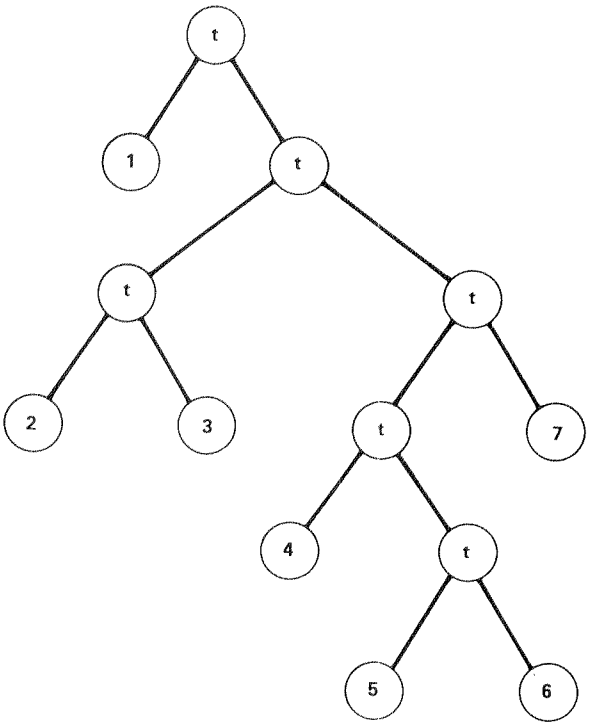


Figure 7–1 The term **t(1, t(tt2, 3), t(tt4, t(5, 6)), 7))** as a tree.

Note the `[H|T]` in the second clause. This unifies with any nonempty list: `H` is set to the first element of the list and `T` to the remainder—the *car* and *cdr* of the list, for those who know Lisp.

Suppose that we call `append/3` with lists as its first two arguments and a variable as its third. Then `append`'s second clause matches, "decapitates" the first list and starts to build the third argument, and then calls itself with the *cdr* of the first argument. It repeats this until the first argument is "[]"; then the first clause matches and the second list is copied into the third argument. As the recursive calls finish, the chopped-off elements of the first list are attached to the third argument, creating what we want. For example, if we append `a,b,c` to `1,2,3`, the successive calls and returns have the following pattern:

```
append([1,2,3],[a,b,c],Z)
        append([2,3],[a,b,c],Z)
                append([3],[a,b,c],Z)
                        append([],[a,b,c],Z)
                        append([],[a,b,c],[a,b,c])
                append([3],[a,b,c],[3,a,b,c])
        append([2,3],[a,b,c],[2,3,a,b,c])
append([1,2,3],[a,b,c],[1,2,3,a,b,c])
```

append/3 also shows that input and output arguments are sometimes interchangeable in Prolog. Calling append/3 with two variables and a list gives, on back-tracking, all the combinations of lists that, when appended, would yield the original:

```
?- append(X,Y,[a,b,c]).

X = [a,b,c]
Y = [] ;

X = [a,b]
Y = [c] ;

X = [a]
Y = [b,c] ;

X = []
Y = [a,b,c] ;

no
```

The cut. A call to a predicate might produce many answers, but we may need only one; several clauses may match a goal, but we might want only the results produced by the first. For interactive queries we may choose not to backtrack (we need not use ;), but this is not general enough.

Prolog provides a construct, the *cut* (written "!"), that satisfies these needs. "!" is written as a goal, although it is not in fact one [26]. When Prolog calls a predicate, matches a clause to the calling goal, and encounters a cut within a clause, the cut succeeds at once; however, if backtracking returns to the cut, the clause fails, no later clauses of the predicate are tried, and so the entire call fails. Consider the following predicate:

```
a(b,X):- !, c(X,Y), d(Y).
a(X,Y):- c(Y,X,Z), e(Z).
```

Calling **a/2** with certain arguments (e.g., two uninstantiated variables) ensures that only its first clause is tried: **a/2** will return whatever answer(s) **c/2** and **d/1** allow. Other cases match only the second clause. Without the cut, **a/2** would behave quite differently.

Negation. Prolog is based on a subset of predicate logic; however, one feature of logic not included is the ability to represent negative information directly, i.e., to state that a relationship is false [7]. For instance, there is no negation operator "¬" to let us write

```
¬ girl(fitzwilliam).

¬ (mother(_,Woman), girl(Woman)).
```

or similar useful facts into our family tree example. Allowing negative facts or goals requires a complicated execution strategy that would make Prolog impractically slow [34]. Instead, Prolog assumes the *closed world assumption* [26]—that its database describes its "world" completely. In other words, it considers true anything that can be deduced from its database and it considers false anything that cannot. Querying the family tree, we obtain

```
?- wife(_,cynthia).

no

?- girl(cynthia).

no
```

That is, someone named Cynthia is probably female, but Prolog has no facts about `cynthia` and both queries fail. This is an example of *negation as failure* [34]: anything that cannot be proved is taken as untrue, just as if there were an explicit negative fact stating as much.

We can use negation as failure explicitly in goals, using the *built-in predicate* `not` (sometimes written "\+"). A built-in predicate is part of the Prolog system, often providing a function that cannot be described in Prolog itself; **not,** however, can be defined easily:

```
not(X):- X, !, fail.
not(X).
```

Note the goal `X`: Prolog lets instantiated variables be treated as goals. If `X` succeeds, the always-failing goal `fail` makes the clause fail, and the cut makes `not(X)` itself fail. If `X` fails, the second clause makes `not(X)` succeed.

Disjunction, if-then-else. Prolog allows clauses to be written with explicit disjunctions. For example, the third of these has the same meaning as the first two combined:

```
a(X,Y):- b(X,Z), c(Y,Z).
a(X,Y):- d(X,Z), e(Z,Y).

a(X,Y):- b(X,Z), c(Y,Z); d(X,Z), e(Z,Y).
```

The disjunction ";" is not mere shorthand: the single clause-head saves a unification, and in some cases we can "factor out" calls common to several clauses.

Prolog's *if-then-else* statement combines the effects of the cut and the disjunction, behaving as if defined by

```
(X -> Y ; Z):- X, !, Y.
(X -> Y ; Z):- Z.
```

The *else* is optional:

```
(X -> Y):- X, !, Y.
```

`(X -> Y ; Z)` chooses `Y` if `X` succeeds and `Z` if it fails.

Dynamic predicates. Prolog allows its database of clauses to be changed during execution. Facts or rules can be *asserted* and *retracted* using built-in predicates. `asserta/1` takes a fact or clause as an argument, makes it the first of its predicates, and succeeds; and `assertz/1` does the same, making it the last. (*a* is the first letter of the alphabet, *z* the last.) For example, `girl/1` has two facts:

```
?- girl (Girl).

Girl = kitty ;

Girl = mary ;

no
```

Now we add a new fact as its last and try it again:

```
?- assertz(girl(georgiana)), girl(Girl).

Girl = kitty ;

Girl = mary ;

Girl = georgiana ;

no
```

As the example shows, the assertion predicates do not succeed again on backtracking.

`retract/1` removes clauses from the database. It tries to unify its argument with the head of a clause of the appropriate predicate, just as with a predicate call. It removes the first clause with a head that matches. On backtracking, it tries to match and remove another clause, failing when no clauses match. It can delete an entire predicate, as seen in the following:

```
?- retract(girl(X)).

X = kitty ;

X = mary ;

X = georgiana ;
```

```
no
?- girl(X).

no
```

Input and output. Prolog has a collection of built-in predicates which perform input and output. They can open and close files, read or write single characters or Prolog terms, and so forth. Like the cut and the assertion and retraction predicates just described, these I/O predicates have *side-effects*: they perform actions unrelated to the logical meaning of a program [45]. A goal that prints a term, for instance, produces a visible result but does not affect the success of its clause.

`read/1` reads Prolog terms and `write/1` writes them; `get/1` and `put/1` do the same for characters. Note that `read` and `put` try to unify their arguments with their inputs, and fail when they cannot. `see/1` and `tell/1` open files for input or output, which are by default interactive; `seen` and `told` close them. Other I/O predicates, varying with the dialect of Prolog, exist as well. An example is:

```
double:- read(Term), write(Term), tab(4), write(Term), nl.

?- double.
|: knock.
knock    knock

yes
```

The system prompts for an input, which for `read` must be a Prolog term followed by a period (`get` is not so choosy). It writes the term, writes four spaces (`tab/1` writes the number of spaces given by its argument), writes the term again, and then goes to a new line.

Metalogical features. Prolog also has built-in predicates that operate on terms and clauses, finding their properties, extracting their components, or building new terms from old. Those that test properties succeed if the property holds. `var/1`, for example, succeeds if its argument is a free variable:

```
?- var(X).

X = _0 ;

no
?- var(not_a_variable).

no
```

(The `_0` is the variable's identifier or address, which depends on the implementation.)

Other built-in predicates build or decompose terms. `functor/3` returns the name and arity of a term, or, given a name and an arity, it builds a new term with new variables for arguments:

```
?- functor(a(b,c,d),F,N).

F = a
N = 3 ;

no
?- functor(X,a_const,5).

X = a_const(_6,_7,_8,_9,_10) ;

no
```

arg/3 extracts a given argument of a term, but fails if the "index" number is out of range:

```
?- arg(2,a(b,c(d,e),f),Comp).

Comp = c(d,e) ;

no
?- arg(5,a(b,c(d,e),f),Comp).

no
```

The predicate "=. .", pronounced *univ* [7], translates between terms and lists; the functor of the term corresponds to the *car* of the list; the arguments, to the *cdr*.

```
?- a(b,c(d,e),f) =.. List.

List = [a,b,c(d,e),f] ;

no
?- Term =.. [t,2,4,6,8].

Term = t(2,4,6,8) ;

no
```

`clause` and interpreters. The built-in predicate `clause/2` is especially useful for writing shells [45]. It allows a Prolog program to access its own clauses. Prolog tries to unify the first argument of `clause` with successive clauses of the appropriate predicate and, when it can, then tries to unify the second argument with the body of the clause. For instance, given the clauses

```
a(X,Y):-  b(X), c(Y,Z), d(Z,X).
a(c,b):-  b(X), e(X).
a(X,Y):-  c(X,Y).
a(X,X).
```

this call behaves as follows:

```
?- clause(a(c,c),Body).

Body = b(c),c(c,15),d(15,c) ;

Body = c(c,c) ;

Body = true ;

no
```

Note that the second clause does not match and that the fact has the trivial goal **true** for a body.

An *interpreter* is a program that executes a language directly, perhaps translating it into an internal representation. It treats a program as a series of commands that change its state—rather like a high-level software analog of a computer. Many expert systems languages are still interpreted [24], and early implementations of Prolog [41] were all interpreters written in conventional languages such as Fortran or C. Prolog is now usually compiled, because that yields faster execution, typically five to ten times faster [9] for a given machine.

For certain applications, such as debugging, providing explanations, or experimenting with different strategies of executing a language, interpreters are still useful. They are easy to design and modify, and are often fast enough for interactive work [45]. `clause` lets us write a Prolog interpreter in Prolog itself:

```
solve(true).
solve((A,B)):- solve(A), solve(B).
solve(A):- clause(A,B), solve(B).
```

This looks trivial—but it can execute most programs that do not use built-in predicates. It is easily modified to handle full Prolog, act as a debugger, or support features needed for an expert system shell [45].

THE WARREN ABSTRACT MACHINE

Introduction

Most introductions to Prolog machines [23,27] describe in detail how a Prolog interpreter (written in a conventional language) works: how it handles forward execution, backtracking, control constructs, and the rest. Only then do they

describe the *Warren abstract machine (WAM)* [54], on which most implementations of Prolog are based nowadays, and the progenitor of most parallel Prolog machines. We describe the WAM and its behavior directly, rather than explain things through an interpreter; the operation of the machine is no more difficult to understand on it own, and we spare the reader possible confusion between the two methods.

The WAM is an abstract machine with data areas, registers, and an instruction set tailored to execute Prolog. Warren devised its original version [53] to describe an intermediate language for his Prolog compiler. The compiler did two translations: from Prolog to the intermediate language, and then from the intermediate language to executable code for the DEC-10 computer. Later Warren altered the WAM [54] so that it could be implemented as a real machine. Many recent compilers [1,4,9] produce WAM code, called *byte-code* because of its terse instructions; this then runs on a WAM emulated in software or microcode, or built as hardware. Byte-code instructions range from simple to complex, from moving data to calling predicates.

Parts of the WAM

Figure 7–2 shows the data areas and registers of the WAM. Note that the contents of a register, say **Q**, are indicated by **(Q)**.

Data areas and their contents. For data, the WAM has five areas of memory. The *code area* contains the byte-code version of the program being executed; this does not change unless clauses are asserted or retracted. (We assume that programs are static: handling of dynamic predicates varies greatly among implementations, and we shall not discuss the details.) The *PDL* (*p*ush-*d*own *l*ist) aids unification; normally it would be called a *stack*, but that word has another meaning in the WAM.

The other data areas are known as *stacks:* the (*local*) *stack*, the *heap* (*global stack*), and the *trail*. The heap contains compound terms (structures), the trail records how unifications bind variables, and the stack keeps track of the progress of execution.

A *symbol table*, not considered part of the WAM proper, contains symbols and arities for each functor (including alphanumeric constants) in the program, so that we can refer to each one by a pointer to its entry. New symbols that appear in data can be added to this table as a program runs (again, in ways varying among implementations).

Data and the heap. *Tags* let us tell variables, constants, structures, and lists apart readily. For a WAM emulated on a conventional machine, two or three bits of a 32-bit word should suffice; for a hardware WAM, longer tags might be more practical. For our examples [9], we assume a 32-bit word, with the two least significant bits forming the tag. (The machine addresses eight-bit bytes, and all data are 32-bit aligned words, so that the two least significant bits are superfluous for addressing.) An unbound (*free*) variable has tag *00* and contains its own address. A

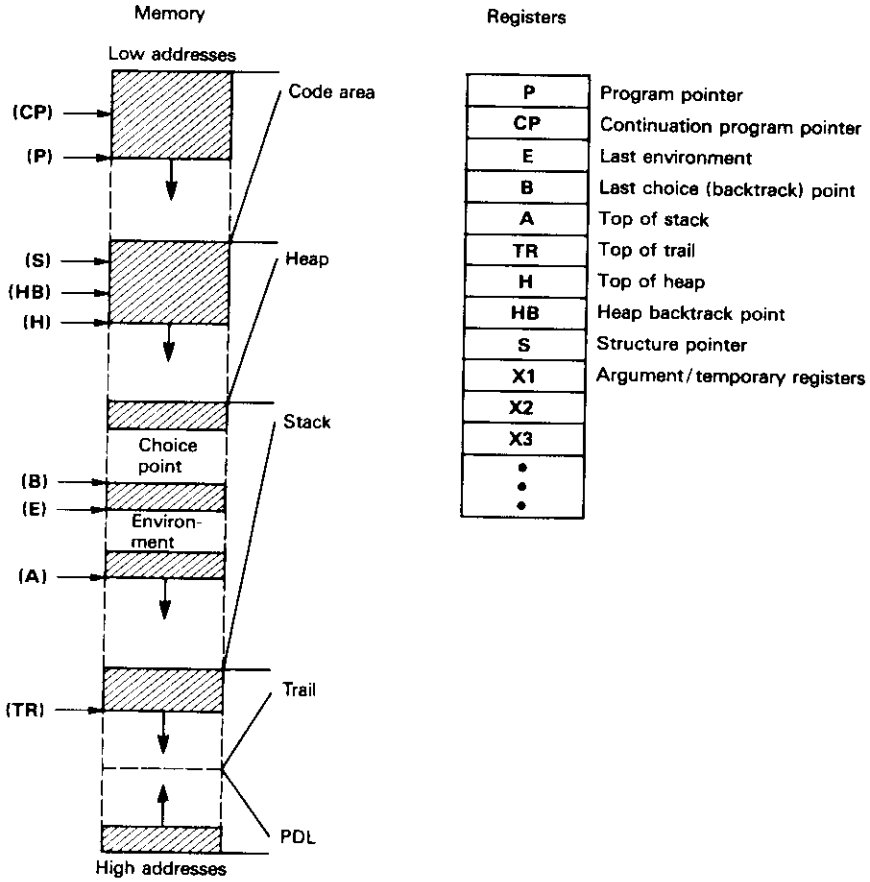


Figure 7–2 Data areas and registers of the WAM.

functor or constant has tag *01* and points to an entry in the symbol table. A number is a 30-bit integer with the tag *10;* it contains its own value. We can represent floating-point numbers (though of low precision) by using a third bit to distinguish between them and integers. A list has tag *11*, except for "[]", which is a constant.

When a program builds a data structure, it builds it atop the heap, with the WAM register **H** keeping track of where the top is. A structure with n arguments uses $n + 1$ contiguous words on the heap: the functor followed by its arguments. Figure 7–3 shows a complex structure on the heap. Free variables and structures appear as names, although in fact free variables point to themselves and constants and structures point to the symbol table. Note the terse representation of lists: for a pointer to a list, the *car* is the word pointed to, and the *cdr* is its successor.

One more point about data representation. We write of a variable or a struc-

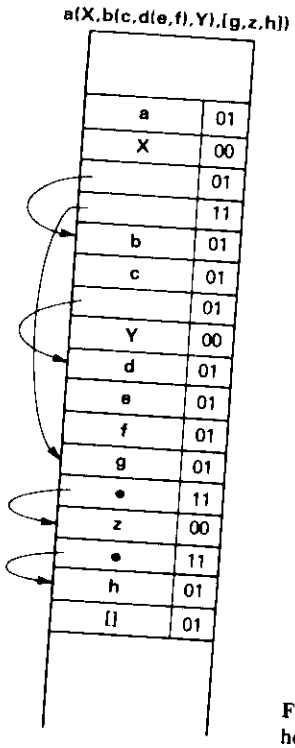


Figure 7–3 A complex structure on the heap.

ture being "stored in" or "copied to" a certain location on the heap or stack or in a register. In many cases we mean that a pointer to it is put there, or possibly even a pointer to a pointer. Going down a chain of pointers to reach the actual variable cell or datum is known as *dereferencing* [27]. Figure 7–4 shows an example: a register **X1** points to a position on the stack which points to an unbound variable on the heap which points to itself. In dereferencing, the WAM "chases" the string of pointers until it finds the unbound variable, which it can then copy into **X1,** making it point to exactly the right location. (Dereference chains are very rarely [51] longer than this.) In describing the operation of the WAM, we shall sometimes mention that dereferencing occurs, but we shall not give details.

Execution and the stack. Prolog's mechanism of execution requires that particular information be saved about the operations performed in reaching the current state—what one author [23] calls "the evolution of the locus of control." The stack does this by storing *environments* and *choice points.*

An environment [54] is pushed onto the stack when a calling goal tries a new clause of a predicate, provided that the clause is a rule with at least two goals. (In the section on registers and variables we show why rules of one goal and facts need

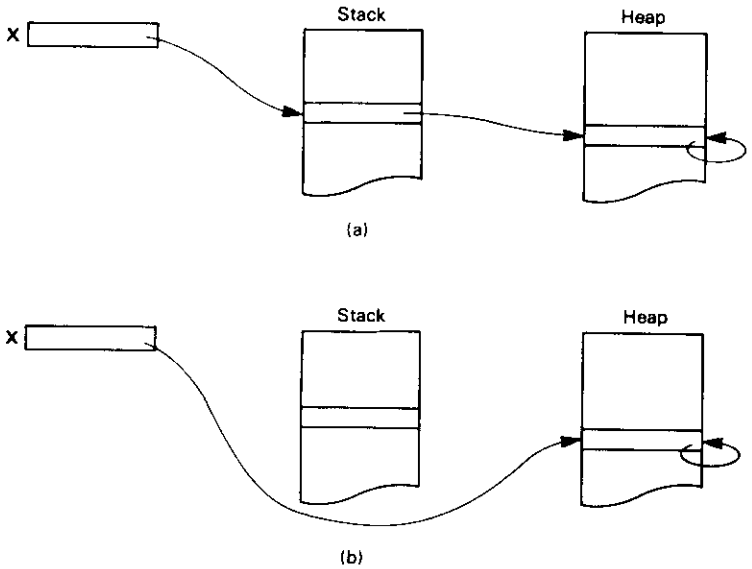


Figure 7–4 Dereferencing (a) Stack and heap before dereferencing. (b) Stack and heap after dereferencing.

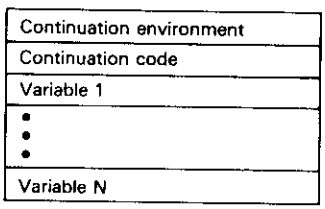


Figure 7–5 An environment.

no environment.) The environment contains a cell for each variable in the clause (again, there are exceptions), and a *continuation*. This is a pair of pointers, one to the code that should be executed after the calling goal succeeds (like a return address for a subroutine call), and the other to the environment already created for that code. Figure 7–5 shows the layout of an environment.

A choice point [54] is pushed onto the stack whenever the clause being tried is the first of a predicate that has more than one clause. It contains the current values of certain machine registers (**H, TR, B, CP, E,** and **X**) sufficient to restore the current state of computation, and a pointer to the next clause of the predicate. When a goal fails, the WAM restores its previous state from the most recent choice point and resumes execution with the next clause. Restoring the old state lets the recent information on the stack and structures on the heap be overwritten, but it doesn't undo instantiations made to extant variables. (That is the purpose of the trail.) Figure 7–6 shows the layout of a choice point.

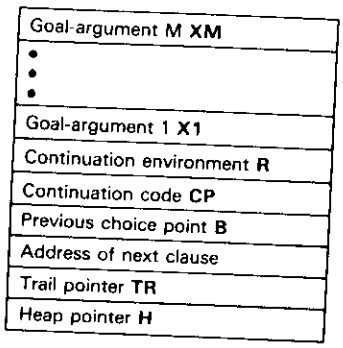


Figure 7–6 A choice point, with corresponding registers.

Instantiations and the trail. Execution instantiates variables; thus, when backtracking discards the results of execution, some instantiations must be undone. When a goal fails, any new variables created by it are discarded automatically, but any variables that already existed and were free when it was called must be free again after it fails.

The problem is easy to solve: we record atop the trail the addresses of variable cells (on the stack or the heap) as they are bound. At every choice point we store the current value of **TR**, which points to the top of the trail. If the calling goal cannot be satisfied by the current clause, we clear every variable cell whose address appears on the trail later than the saved **TR**, and reset **TR** from the saved copy.

Registers and variables. The WAM has a set of registers which point into the data areas, and whose values are saved in choice points [54]. (See Figure 7–2.) **P**, the *program pointer*, points to the current instruction of the program. **CP**, the *continuation pointer*, holds the return address for the current predicate call—i.e., the location of the code to execute next (e.g., the next goal) if the current goal succeeds. **E** points to the *last environment*, letting us access the variables and continuation information stored in it. **B** points to the last choice point (point of *backtrack*), so that the WAM can restore its previous state and try any alternative clauses for the predicate being called. **A** points to the top of the stack, so that the WAM knows where to put new environments; **A** is not essential—the top of the stack is an offset from **E** or **B**—but it makes execution faster.

We have already seen **TR**, the top of the trail, and **H**, the top of the heap. **HB**, the *heap backtrack* point, holds the value of H at the time the last choice point (pointed to by **B**) was created. **S** points to existing structures on the heap, allowing access to their arguments. There is also a set of registers that hold both temporary variables of a clause and the arguments of a goal being called. Warren [54] calls these by different names when they are used for different purposes: **X** registers (**X1**, **X2**, and so on) for temporary variables, **A** registers (**A1**, **A2**, etc.) for arguments. We follow recent authors [16,27] and compilers and always call them **X** registers henceforth.

A *temporary variable* [54] occurs first in the head of a clause, in a structure, or

in the last goal, but not in more than one goal of the body, counting the head as part of the first goal. Any condition in this seemingly peculiar set indicates that the variable exists elsewhere or will be needed only briefly, so that it will not have to be stored in an environment. A *permanent variable*, denoted **Y1, Y2,** and so forth, is any that is not temporary; it must be stored in an environment, where a cell is reserved for it, and the WAM addresses it by an offset from the pointer to its environment. Environments are arranged such that their variables' cells have the highest addresses. If the compiler puts variables in a suitable order, the top addresses of an environment can be *trimmed* away and used for the next stack item when the corresponding variables are no longer needed.

The WAM Instruction Set

WAM instructions fall into five classes: **get, put, unify, procedural,** and **indexing.** The entire set proposed by Warren is in Table 7–1, but we discuss each class separately, basing our descriptions on his [54]. Implementors [9,51] have extended the set, renamed some instructions, and so forth, but Warren's design remains the standard, and we use his names.

	Head	Body
Procedural	proceed	execute P call P, N deallocate
Get/put	get_variable Xn, Ai get_variable Yn, Ai get_value Xn, Ai get_value Yn, Ai get_constant constant, Ai get_nil Ai get_structure functor, Ai get_list Ai	put_variable Xn, Ai put_variable Yn, Ai put_unsafe_value Yn, Ai put_constant constant, Ai put_nil Ai put_structure functor, Ai put_list Ai
Unification	unify_void N unify_variable Xn unify_variable Yn unify_local_value Xn unify_local_value Yn unify_value Xn unify_value Yn unify_constant constant unify_nil	
Indexing	try_me_else label retry_me_else label trust_me_else fail switch_on_term var con lis str switch_on_constant N, table switch_on_structure N, table	try label retry label trust label

Table 7–1 WAM instructions.

`get` instructions. A compiler generates a **get** instruction for each argument of a clause's head. They match a variable that is definitely uninstantiated (**get_variable**), a variable that might be instantiated (**get_value**), a constant (**get_constant**), a structure (**get_structure**), or a list (**get_nil, get_list**) to an argument (already copied to an **X** register with a **put** instruction) of a calling goal. A variable can be temporary (in an **X** register, so that the **get** moves it from one register to another) or permanent (in a slot **Y** of an environment). Note that the list instructions have only one argument: the functor or constant is known.

`put` instructions. **Put** instructions load an argument of a goal into an **X** register, where a **get** can retrieve it. Each **put** is the analog of the **get** of similar name, except for **put_unsafe_value.** An *unsafe* variable does not first occur in the head or in a structure, and is therefore set by a **put_variable. put_unsafe_value** is used instead of **put_value** for the last goal in which an unsafe variable appears. Why?

Suppose that the variable is uninstantiated. A cell for an uninstantiated variable points to itself, so that **put_value** copies a pointer to the variable into an **X** register. Now the portion of the environment containing the variable cells will be trimmed by the **execute** or **call** instruction executed presently. So unless the WAM instead reserves a cell on the heap for the variable and loads a pointer to it into the **X** register, that **X** will point to something else.

`unify` instructions. A compiler generates a **unify** instruction for each argument of a structure (or list). A sequence of **unify**s is always preceded by a **get** or **put** of a structure or list. If the structure already exists, unification is between the argument of the instruction and the corresponding argument of the structure. The **S** register points to the structure, and is incremented for each successive argument. If the structure is being built, the **unify**s construct its new arguments atop the heap from their own arguments, updating the top-of-heap register **H.**

unify_void N unifies some number **N** of anonymous (single-occurrence) variables, which require no variable cells. **unify_local_value** replaces **unify_value** if the variable has not been initialized to something on the heap (a *global* value).

Procedural instructions. Procedural instructions handle the overheads of calling goals in the body of a clause and manage the stack space used by the environment. A fact compiles into **get**s (and possibly **unify**s), followed by **proceed,** which simply transfers control. A rule requires that space for an environment be **allocate**d on the stack. After that, the WAM must **get** the arguments of the head. For each goal, the arguments are **put** into the **A** registers, and the appropriate predicate is **call**ed. The second argument of **call** is the number of variables in the environment; initially there is a cell for each variable, but some can be discarded as the clause executes. The last goal of a clause needs nothing from the environment (recall the **put_unsafe_value** instruction that helps ensure this), so that the environment is **deallocate**d (removed) from the stack and the goal is called using **execute.**

Note that a rule with one goal needs no environment: **allocate** and **deallocate** are not used, and the goal is **executed.**

Indexing instructions. Naive implementations of Prolog [38,41] try to match a call with every clause of the appropriate predicate. Often this is a waste of time. Consider the following example:

```
a([],[]).
a([p(X,Y)|T],U):- . . . .
a([q(X,Y)|T],[r(Y,Z)|U]):- . . . .
```

Suppose that we call a/2 with a (nonempty) list as its first argument. Then trying the first clause is useless. Similarly, if the first argument is indeed the empty list, trying the other clauses is useless.

Indexing instructions are meant to keep the WAM from trying clauses that cannot match a call. They use the principal functor of the predicate-call's first argument (in register **X1**) as a *key* to decide which clauses to try. If **X1** dereferences to a variable, the WAM must try every clause. In order to handle this case, a **try_me_else** instruction precedes the code for the first clause, a **retry_me_else** that for intermediate clauses, and a **trust_me_else fail** that for the last clause. The operand is the address of the next clause; the **fail** of **trust_me_else** shows that no alternatives remain, and the call fails if this clause fails.

switch_on_term examines **X1.** Its four arguments are the addresses to branch to if the dereferenced **X1** is a variable, constant, list, or structure. The first is the address of the predicate's **try_me_else**; the others are addresses either of the code for a single matching clause or of that of a block of matching clauses. Such blocks have **try**, **retry**, and **trust** instructions before their initial, middle, and final clauses, respectively, analogous to the **try_me_else** instructions for the variable case. If diverse constants or structures appear in the first arguments of clauses, **switch_on_constant** and **switch_on_functor** look in hash tables to find the addresses of the code for each case.

Examples of Execution: append/3

The foregoing may seem cryptic, so in this section we present several examples to show that the operation of the WAM is actually quite straightforward. For each example, we give Prolog code, the corresponding WAM "assembly language," and descriptions of what happens in the WAM in response to various calls.

Earlier we presented append/3:

```
append([],L,L).
append([H|T],L,[H|T2]):- append(T,L,T2).
```

Compiled into WAM assembler, it looks like this:

```
procedure append/3
    switch_on_term _580,_578,_579,fail      /*var.,cont.,list,struct.*/
_580:
    try_me_else _581,3                      /*var.*/
_578:                                       /*create a choice point*/
    get_value X2,X3                         /*const.*/
    get_nil X1
    proceed
_581:
    trust_me_else fail                      /*discard the choice point*/
_579:                                       /*list*/
    get_list X1
    unify_variable X4
    unify_variable X1
    get_list X3
    unify_value X4
    unify_value X3
    execute append/3
```

(This is output from a real compiler [51], slightly doctored to conform to the original WAM code.) The code demonstrates many things presented in the previous section. With a variable in the first argument, **switch_on_term** branches to the first label so that both clauses are tried; with a constant, it branches directly to the first clause; with a list, to the second; and with a structure, to failure, because neither clause head can match a call that has one in that position. The first clause is a fact, so that its empty body is "called" using **proceed**; the second has only one goal, so that no environment is **allocate**d (or, clearly, **deallocate**d); and the recursive call uses an **execute.**

A deterministic call. Let us follow the operation of the WAM as it executes the goal

```
append([a,b,c],[1,2,3],X).
```

Figure 7–7 shows the initial state of the registers and data areas. Note that the first three X registers contain the arguments of the call, in this case (pointers to) two lists on the heap and one variable of the current environment.

Now the WAM calls `append`. The **switch_on_term** sees that **X1** is a list (by checking its tag) and branches to the label **_579,** the code for the second clause. All the **try** instructions are avoided, because no choice point needs to be created: `append` is deterministic in this mode, returning only one answer. **get_list** dereferences the first argument, finds that it is a list, not a variable, and puts the WAM into *read mode.* This means that **S,** the register that points to structures, is set to the first argument of the list, and that subsequent **unify** instructions will try to match the existing list rather than build a new one.

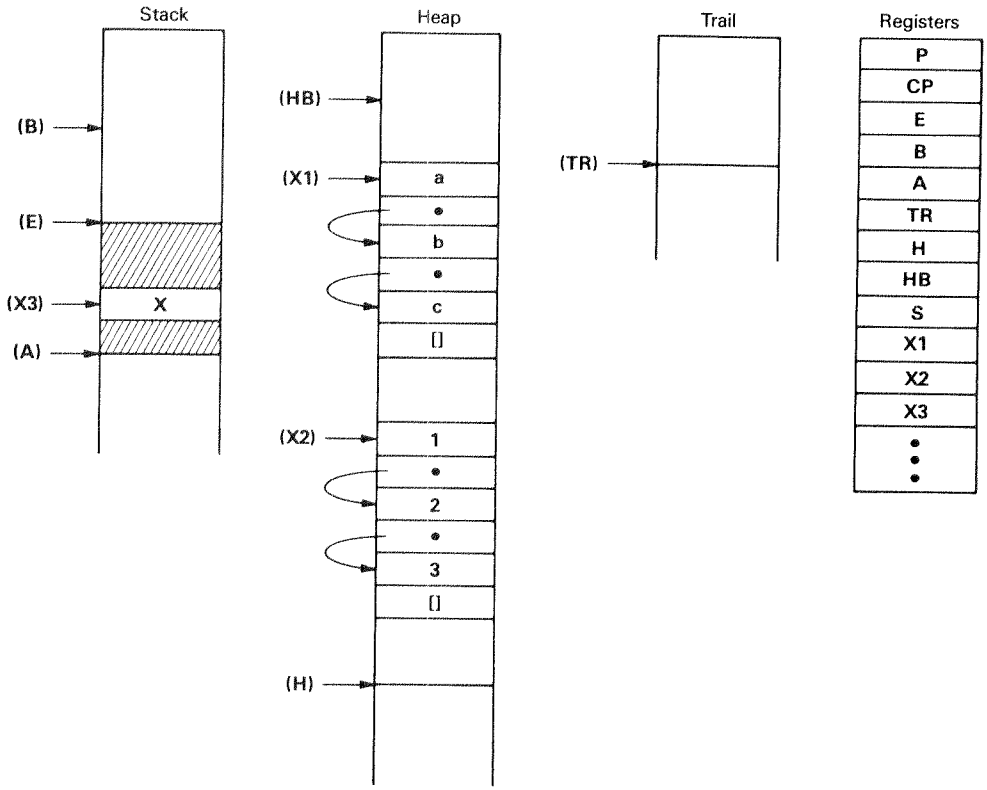


Figure 7–7 Just before calling **append([a,b,c],[1,2,3],X).**

The first **unify_variable** copies a, the head of the list, into **X4** and increments **S** to point to the next argument, [b,c] (represented as .(b,.(c,[]))), the tail of the list. The next **unify_variable** then gets .(b,.(c,[])) into **X1,** which is also the first argument register for the next call. (The compiler is being clever here.) The second argument is passed unchanged to the call, so it is left alone.

The third argument, **X3,** is a variable; **get_list** therefore puts the WAM into *write mode.* Instead of addressing with **S** a list extant on the heap, the WAM copies **H** to **X3,** giving the copy a list tag. The **unify_value** then sets the head of the new list (atop the heap) to **a,** and the **unify_variable** makes the tail a free variable (untagged pointer to itself) and copies it to **X3** to prepare for the call. Now **X1** has the tail of the first list, **X2** the untouched second list, and **X3** the variable tail of the list being built, as shown in Figure 7–8. The WAM now makes the recursive call to **append,** using **execute** because there is no environment to maintain.

The recursive call proceeds much as the first one did. So does the next one. Finally the list is whittled down, and we call **append** with [ ] as its first argument. **switch_on_term** then branches to label **_578** and the code for the first clause. Next,

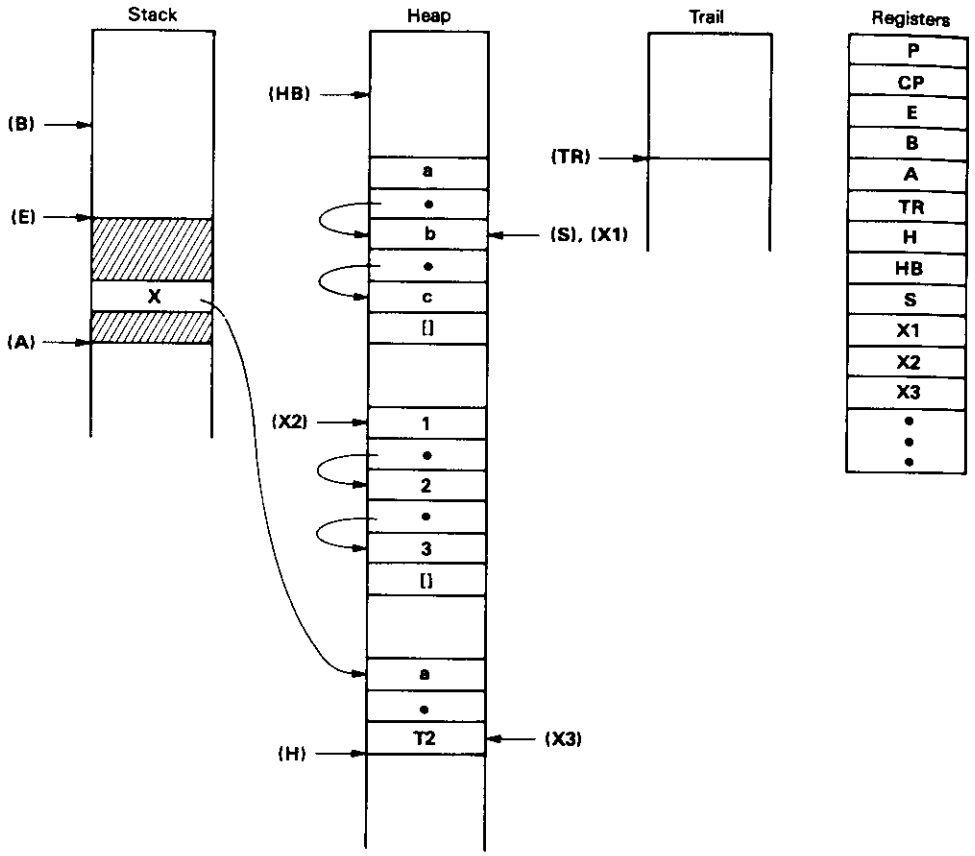


Figure 7–8 Just before the first recursive call.

get_value sets the variable atop the heap (pointed to by **X3**) to point to the list pointed to by **X2**, forming the "append" of the lists. Now the **get_nil** matches **X1**, and the **proceed** sets off a series of returns. At the end of these (see Figure 7–9) **X3** points to the new (slightly odd-looking) list atop the heap, and the call to **append** has finished.

A nondeterministic call. Consider the call

```
?- append(X,Y,[a,b]).
```

which generates the three possible pairs of lists that, when appended, yield `[a,b]`. We assume that the **X** registers have been loaded with (pointers to) the two vari-

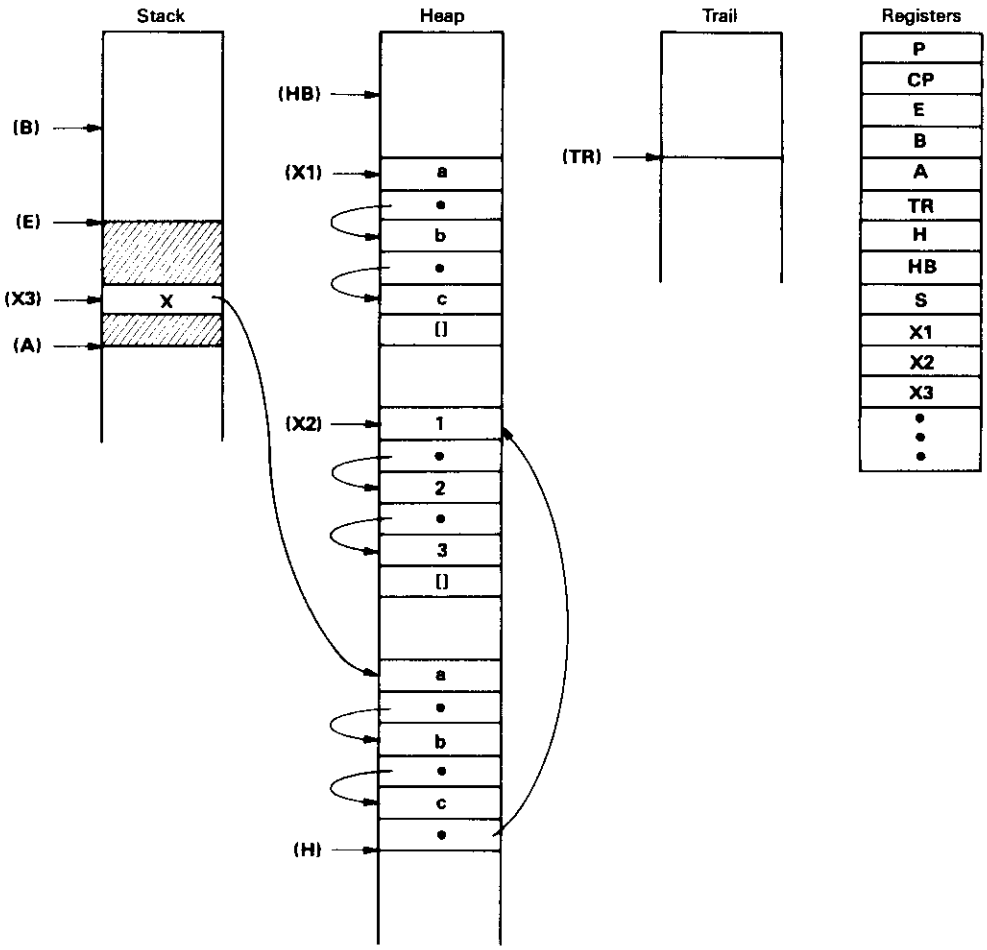


Figure 7–9 Just after the goal completes.

ables and the list, as shown in Figure 7–10. This time **switch_on_term** branches to **_580**: the WAM will try both clauses. The **try_me_else** makes a choice point on the stack, with three slots for the arguments and six more for the state of the machine, viz., **E,** pointing to the current environment; **CP,** the current continuation; **B,** the previous choice point; **L,** the next clause (the argument of the **try_me_else**); **TR,** the top of the trail; and **H,** the top of the heap. The heap-backtrack pointer **HB** is set to the current **H,** and **B** is set to the current top of the stack. That out of the way, the **get_value** copies the list-pointer in **X3** to **X2,** the **get_nil** sets **X1** to `[]`, and **proceed** completes the call. We get `X = []` and `Y = [a, b]`, as shown in Figure 7–11.

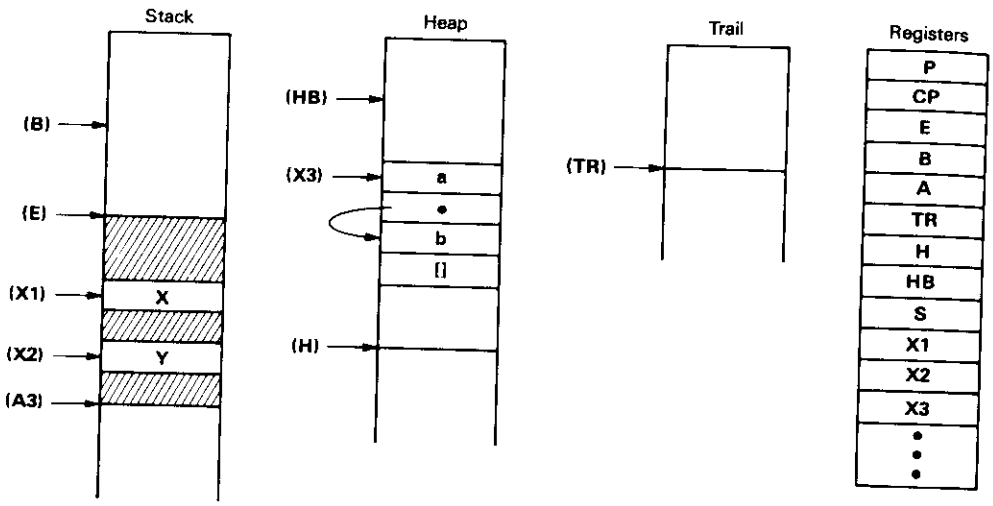


Figure 7–10 Just before calling **append(X,Y,[a,b])**.

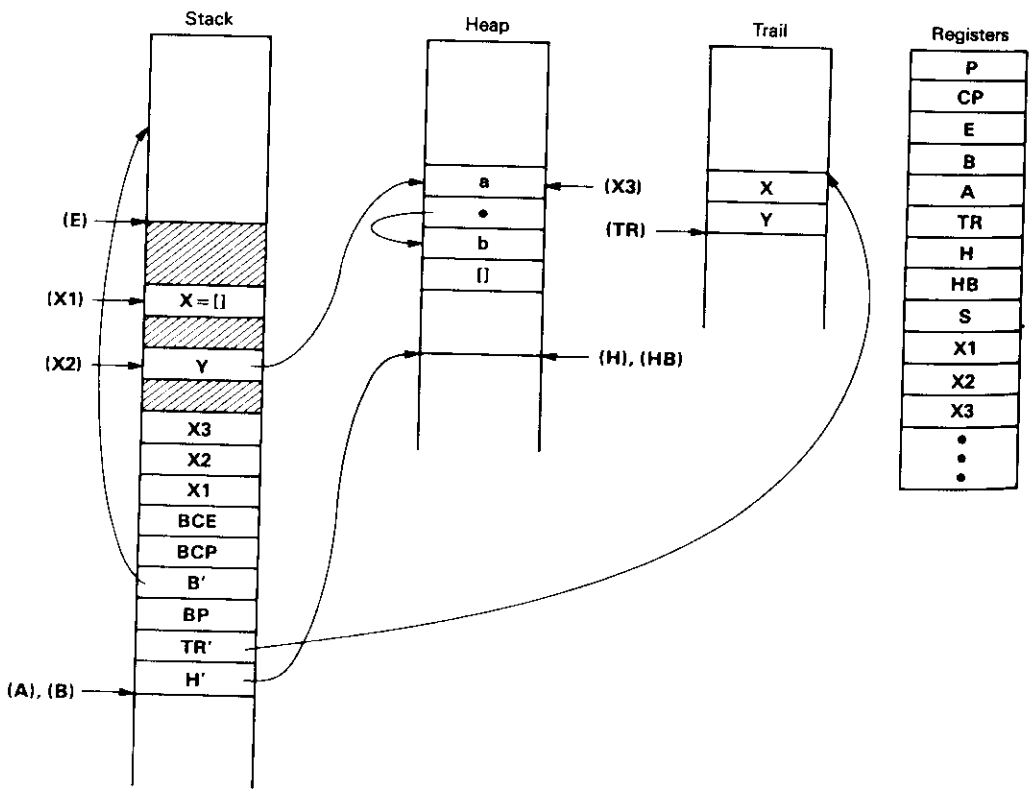


Figure 7–11 First answer.

Now suppose that we try the calling goal again. Then the WAM branches to the label **_581** of the next clause (retrieved from **L** in the choice point). There the **trust_me_else fail** resets registers **B** and **HB** with their values from the previous choice point and then discards the current choice point. This time the first **get_list** gets a variable and puts the WAM into write mode, building a new list atop the heap. Now the **unify_variables** put (pointers to) the new (variable) head in **X4** and the new tail in **X1**. The second **get_list** gets the list `[a,b]`, and the subsequent **unify** instructions make `a` the head of the new list and make **X3** point to `[b]` (Figure 7–12). The WAM then makes the recursive call.

switch_on_term branches to the **try_me_else**, which creates a choice point much as before. The **gets** point the second argument to `[b]` and complete the list on the heap with `[]`. The **proceed** then completes the call, the previous call also completes, and `append` returns `X=[a]` and `Y=[b]` (see Figure 7–13).

If execution again backtracks to `append`, the WAM returns to the most recent choice point, where it gets the label **_581** of the second clause. Again the **trust_me_else fail** discards the choice point and retrieves the **B** and **HB** registers from the previous choice point. Everything happens much as it did the last time this clause was tried; before the recursive call, `b` is added to the list being built on the heap, and **X3** has `[]` (see Figure 7–14).

Again the **try_me_else** creates a choice point. The **gets** then set the second argument to `[]` and match the other `[]` on the heap. When all the calls complete, `append` returns `X=[a,b]` and `Y=[]` (see Figure 7–15).

No more answers are possible. Backtracking again restores the saved state from the choice point and then discards the choice point. The call fails on the **get_list X3** instruction, because `X3` now points to a `[]`.

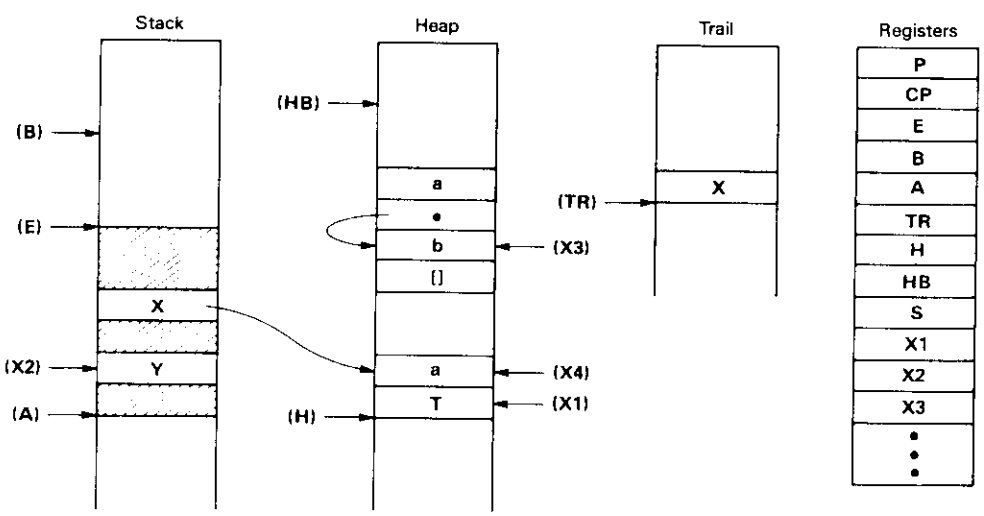


Figure 7–12 Just before the first recursive call.

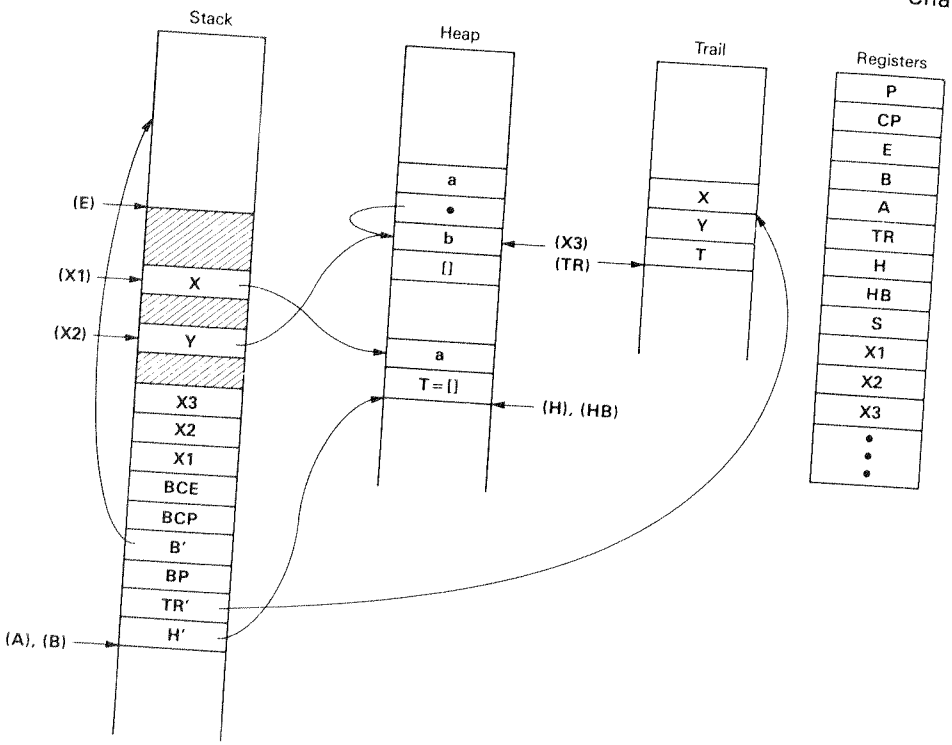


Figure 7–13 Second answer.

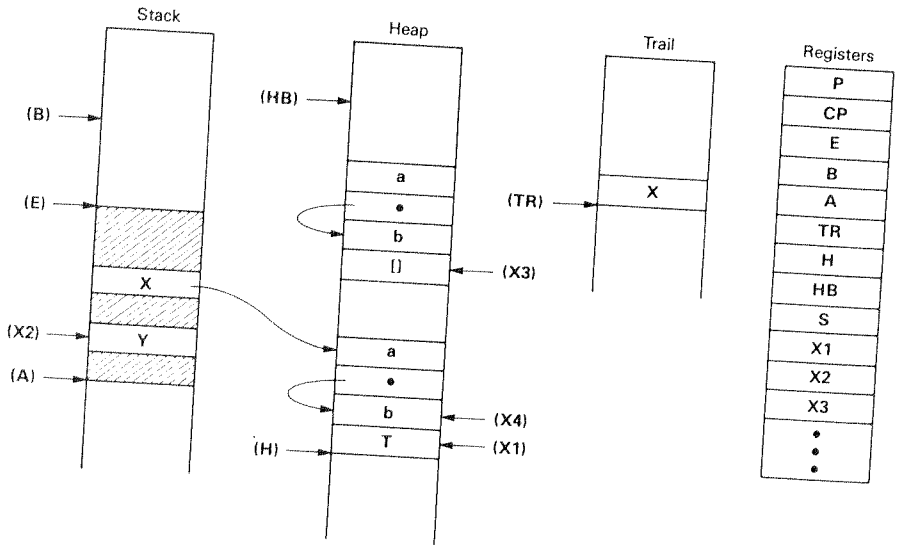


Figure 7–14 Just before the second recursive call.

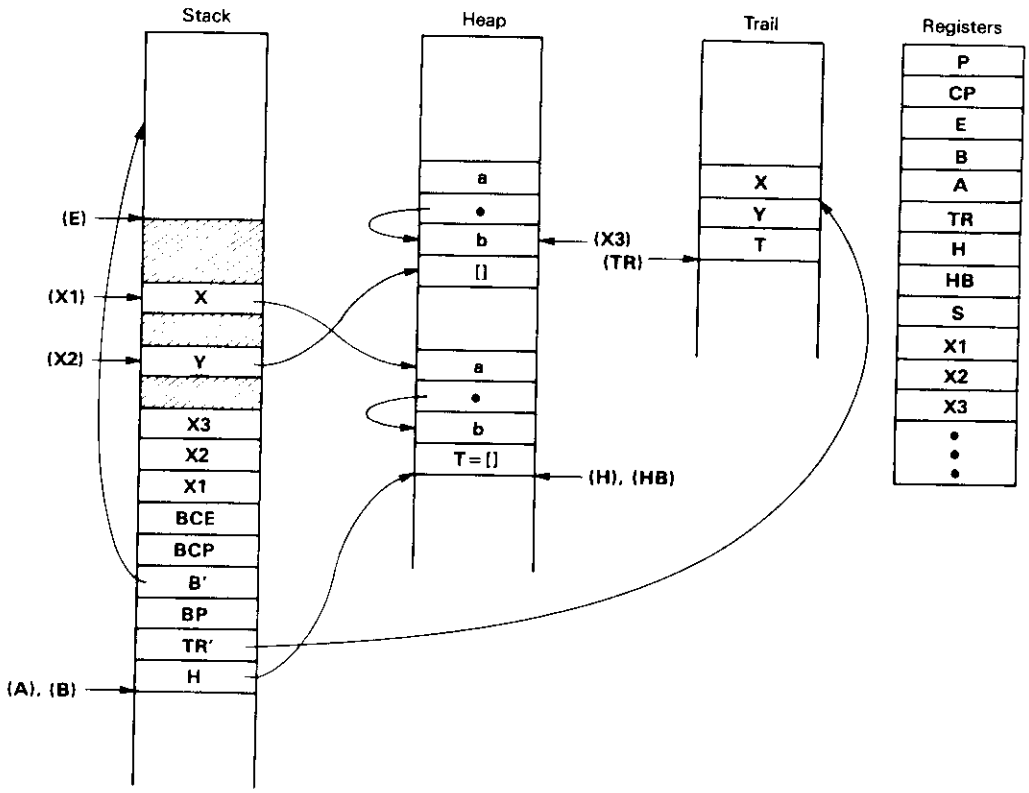


Figure 7–15 Third (final) answer.

Implementing the WAM

Essentially, there are three ways to implement a WAM: in software, microcode, and hardware. Each has its advantages: software is easy to modify and portable, microcode gives some of the speed of hardware but some of the flexibility of software, and hardware gives the best speed of all. On the other hand, software can be slow, microcode cryptic, and hardware inflexible and expensive.

Software. A Prolog compiler can produce WAM instructions, or it can translate them to sequences of instructions native to the machine that it is compiling for. The first Prolog compiler [53], written by Warren for the DEC-10 and still held in high regard, did the latter, producing DEC-10 instructions from an early form of WAM code. Most academic and many commerical [1] compilers produce byte code and run it on a WAM emulator. The speed of such implementations depends on the emulator. Writing in a high-level language provides portability and easy mainte-

nance at the expense of speed; assembly language and threaded code [1] are other possibilities.

Implementors who choose this method must live with the constraints of the underlying machine's architecture. Recall our examples, which are based on the public-domain SB-Prolog system [9] (a good implementation using a software WAM): on a machine with 32-bit words and addresses, we find that a tag larger than two bits costs us access to some of the virtual address space. There is no tidy way to handle floating-point numbers: a tag ruins their precision, accessing them with pointers is slow, using double precision creates a data item of another size. Managing tags is a constant nuisance if our target machine doesn't support them. Also, trying to use machine registers for **X** registers can be difficult; it is easier (but slower) to use memory instead [9].

Microcode. There are several examples in the Prolog literature of WAMs implemented in the microcode of other machines [15,17]. Their execution speeds compare favorably to those of software emulators; however, the microcoder is at the mercy of the machine's architecture and microarchitecture. Much of the gain in performance seems to come from microcoded unification. Special-purpose microcoded machines, with many registers and the ability to handle tags, should fare better than general-purpose machines [15]. With the rise of powerful microprocessors that are hardwired for speed, microcoded machines are less common now than a few years ago.

Hardware. Many researchers have implemented the WAM in hardware [13,25,35,48]. In general, these machines have microcoded control, but of a WAM-style architecture. Typically they have large tags (e.g., a 40-bit word, eight bits of it tag), with some bits used to support "garbage collection." Memory management, dynamic predicates, tag checking, and the so-called garbage collection may all be supported in hardware. Some are stand-alone machines, reminiscent of Lisp machines; others are designed to be attached to a host computer.

PARALLEL MACHINES FOR PROLOG

Comparison: Parallel Machines for Expert Systems

Typically, machines designed to support expert systems do so by supporting production systems, assuming forward-chaining programs written in OPS5 or the like. All of them are parallel, and only one of them [42] consists of specialized processors analogous to the WAM. The others are collections of conventional processors, sometimes connected in a complex way, together with logic that is supposed to help production systems execute. The architecture proper is less important for these systems than the algorithms they run, notably parallel-match algorithms such as parallel Rete [31]. Early designs had about a thousand small processors, but now those with "tens" of processors [33], each one fairly powerful, are fashionable; most

programs, however, lack the parallelism needed to exploit more than a few processors.

DADO [46,47] appears to be the earliest architecture meant expressly for expert systems. Up to several thousand processors, each with a small local memory and some circuitry for communication, are connected as a complete binary tree. A processor operates in SIMD (single instruction, multiple data) mode, executing instructions sent to all its descendents from some processor higher in the tree, or in MIMD (multiple instruction, multiple data) mode, executing instructions from its local memory. The rules of a production system can, for example, be distributed amongst the processors (ideally, each gets one rule), so that data to be matched against stored rules can be broadcast and the matching done in parallel.

DADO is really a general-purpose multiprocessor, as well- or ill-suited to executing production systems as a conventional computer. Diverse applications [47] have been proposed for DADO or run on the prototype machines—Prolog, Lisp, and image-processing software, for example. The largest prototype to date is the DADO2, with 1,023 eight-bit microprocessors.

Several recent machines are, essentially, modern versions of DADO: similar to general-purpose machines, with tens of closely coupled processors. The PSM architectures proposed by Gupta et al. [18] are typical: not more than 64 processors, each with a local memory and cache, with a shared global memory for the lot and a "hardware task scheduler." MANJI [32,33] is similar: a small number of powerful but ordinary microprocessors, this time with special multiport memories divided into pages and equipped with page tables. Each processor therefore has its own virtual memory which can be accessed by other processors at the same time.

Oshisanwo and Dasiewicz [37] propose a complicated model for parallel execution of production systems and then base an architecture on it. They use tens of identical processors, grouped into three very different modules that correspond to three parts of their model. One module is a DADO-style binary tree, another a collection of independent processors, and the third a collection of four-processor clusters.

PESA-1 [42] is the closest in spirit to the parallel Prolog machines we describe, in that its processors have a specialized instruction set, analogous to the WAM's. However, these processors are arranged into a pipeline, each stage being a row of processors sandwiched between two buses. Each bus serves as an input for one row and an output for the next, so that the pipeline forms a ring of alternating buses and processor-rows. Buses can be reconfigured to vary the number of stages or the number of processors in a stage, although it is not clear at what times this is done or how the machine's new configuration is decided on.

Prolog Machines

Parallel architectures for Prolog usually exploit *OR-parallelism*, *AND-parallelism*, or both. Usually they are collections of modified WAMs; experimental machines are often ordinary multiprocessors with each processor emulating a WAM. We de-

scribe first the different forms of parallelism (basing our classification on Conery's) and then some representative architectures and models of execution.

Forms of parallelism in Prolog. OR-parallelism [8,55] is the parallel execution of the clauses that match a call to a predicate. Each clause provides some number of answers to the call, so that answers to a call are found in parallel. AND-parallelism [8] is the parallel execution of the goals in a clause. Each goal provides part of any solution to a clause, so that parts of a solution to a clause are found in parallel. Most other forms of parallelism are *low-level* [8], dealing with handling terms (as in parallel unification) or WAM instructions (as with a pipelined WAM), rather than larger objects such as goals or clauses.

Forms of OR-parallelism. Conery [8] describes three types of OR-parallelism: *pure, OR-process,* and *search.* Pure OR-parallelism spawns a new process for every alternative clause; the process reports success (and the bindings it made) or failure, and then terminates. This tends to create a huge number of short-lived processes which do only simple computations and communicate a great deal, not a very practical design. The OR-parallelism used in recent machines (described in a later section) can be considered a restricted form of pure OR-parallelism. OR-processes resemble the objects of an object-oriented language: each is responsible for perhaps a few clauses or predicates of a program, and they communicate by passing messages. Search parallelism results from partitioning a program, distributing its clauses amongst processors. Typically this requires a program with many clauses per predicate, so that multiple processors work at matching a goal to one of these clauses.

Forms of AND-parallelism. *Stream parallelism* treats a variable common to two goals as a channel of communication between them; for example, the first goal can build a list element by element, and the second can process each element of the list as it becomes available. This is a major source of parallelism in parallel languages related to Prolog, such as Concurrent Prolog [43], Parlog [6], and GHC [52]; users of these languages must specify it explicitly, annotating variables suitably. *AND-processors* try to solve goals of a clause simultaneously. In this case, variables shared between goals are a nuisance, because two goals running simultaneously may bind a variable inconsistently. We discuss two solutions to this shortly.

Other forms of parallelism. *Parallel unification* [40] has already been considered; however, it seems that most unifications are trivial, that unification is too low-level to make communication practical between many processors, and that there is a theoretical limit on the speedup [14] (roughly logarithmic in the number of processors). Pipelining the WAM provides parallelism of a sort [49], but it does not appear to be effective [50]. Various combinations of OR- and AND-parallelism have been proposed [58].

Some parallel models and architectures for Prolog. Early parallel architectures for Prolog resemble the general-purpose multiprocessors for production

systems: the architecture is not as important as the algorithms that run on it. Most of the designs we present are more recent. As with the production-system machines and with multiprocessors in general, these have tens of processors rather than hundreds or thousands, and usually some form of shared global memory.

Conery's AND/OR-Process Model (AOPM). Conery [8] does not specify an architecture but assumes a set of processes that communicate by message-passing; a wide range of architectures can support this model of execution. Loosely speaking, his *AND/OR-Process Model* (AOPM) has two types of process. An *AND-process* manages a conjunction of goals, such as a compound query or the body of a clause, deciding which of its goals to evaluate next. It creates, for each selected goal, an OR-process that produces solutions for the goal. An *OR-process* will generally spawn descendent AND processes in turn, and so forth.

An OR-process finds all the clauses that match its goal. It creates an AND-process for each rule, keeping a list, but handles facts on its own. The OR-process sends its first answer (obtained by either means) to its parent AND-process and *gathers* any subsequent results. The parent might send a *redo* message, asking for another answer. If the OR-process has any, it returns one; if not, it sends redo messages to those of its descendent AND-processes that are still active. If all descendent ANDs are finished and the store of answers is exhausted, the OR-process responds to a redo with a *fail* message. When the parent AND fails (due to the failure of one of its ORs), it sends a *cancel* message to all its descendent ORs, which they in turn propagate to their ANDs. The message reaches the entire tree of processes, which terminates, starting from the leaves and progressing upwards.

AND-processes are more troublesome. When a variable occurs in more than one goal, the OR-processes for the goals might instantiate it to different values. Conery solves this by forbidding goals with a shared uninstantiated variable to run in parallel. One instance of a variable is selected (from a mixture of run-time and static information) as the *generator* of that variable's instantiation, with the others being *consumers*. Once the variable is instantiated, goals containing it can safely be executed in parallel.

Backtracking is difficult in the AOPM. As in the WAM or in any Prolog interpreter, we must undo bindings done by goals between the latest backtrack point and the point of failure; in the AOPM, the corresponding OR-processes will have to issue redo messages. Finding the latest backtrack point is complicated but practical.

The RAP Machine. DeGroot's *Restricted AND-parallelism* (RAP) [10,11] resembles the AOPM's AND-parallelism, but relies on compile-time rather than run-time analysis. RAP allows parallel execution only if a few simple run-time tests (e.g., of instantiation and independence of variables) can determine that it will not instantiate variables inconsistently. We lose instances of parallelism, but save run-time overhead. For example, if we have the clause

```
a(X,Y):- b(X), c(X,Y).
```

the compiler generates something equivalent to

```
a(X,Y):-
var(X)->
        b(X), c(X,Y)

        (b(x) & c(X,Y).
```

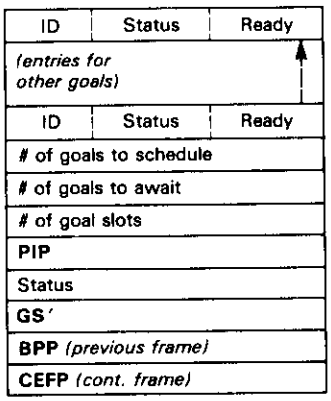
where the & represents parallel execution of goals. Thus, the parallel code runs only when it is safe to do so.

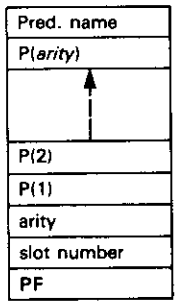
Hermenegildo [20–22] presents the *RAP machine,* a multiprocessor made of modified WAMs. He describes the extra capabilities a WAM needs to support AND-parallelism and then proposes extra hardware and an altered model of execution to provide them. A processor must be able to assign parallel work to other processors and keep track of the state of this work, yet retain the useful features of the original WAM. Hermenegildo shows that his design does these things and supports RAP. Figure 7–16 shows the data areas for one processor of the design and the structure of two new types of record.

Records for parallel calls, called *parcall frames,* now appear on the stack. Each processor also has a new *goal stack* onto which it pushes *goal frames* for goals that are ready to be executed in parallel. Each goal frame contains the information needed for remote execution, viz., a pointer to the predicate being called, a copy of the argument registers for the call, the predicate's arity, a pointer to its "parent" parcall frame (EPF), and its position within that parcall frame. When execution of a clause reaches a parallel call, a goal frame is pushed onto the goal stack for each of the goals, and processors (including the one executing the clause) can "steal" these frames and start to execute the goals.

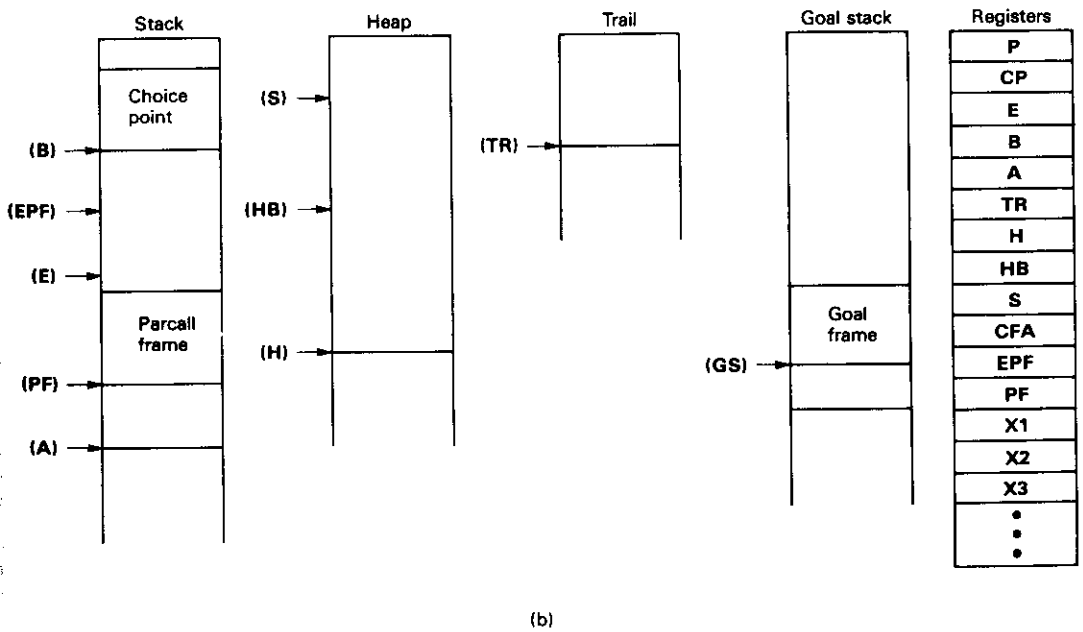
The processor creates a parcall frame on the stack for each parallel call. Within this frame, each goal has a *slot* with three fields: the number of the processor that "stole" it for execution, a bit telling whether it still has alternatives, and a "ready" bit telling whether the goal's frame will actually be put on the goal stack (this is needed when execution backtracks into the parallel set: the retry should involve only goals with alternatives). The frame also holds the number of goals left to schedule, the number executing but not completed, a pointer to the first goal of the parallel call (the *P*ut *I*nstruction *P*ointer, used after backtracking to find parallel goals to be retried), a bit showing whether the parallel call has been backtracked into, the top of the goal stack when the parallel call began, a pointer to the previous parcall frame, and the previous parcall frame pointer (*CEPF*). *PF* is a new register that points to the last parcall frame—the one to try in case of backtracking.

Much of the information in the parcall frame supports backtracking. The scheme works in such manner that in a clause containing a parallel goal, goals can fail in three places: before the parallel goal, within it, and after it. In the first case, we backtrack as usual; in the second, the parallel goal fails and we backtrack to the previous goal; in the third, we backtrack as usual until we reenter the parallel goal. When we do, we consider only the goals that still have alternatives (recall that bit in





(a)



(b)

Figure 7–16 The RAP machine. (a) New data items. (b) Data areas and registers.

a parcall frame's slot). We backtrack among them in the usual way; when one succeeds, we lump those on its right into a smaller parallel goal and execute it (this is the use of the "ready" bits). Of course, if the goals all fail, we backtrack to the goal before the original parallel goal.

Recent OR-parallel Machines. In the past few years various researchers have proposed several execution models for OR-parallelism, closely resembling each other and with corresponding architectures [55,56]. These models consider program execution as building an *OR-tree,* as shown in Figure 7–17. Note that each child of a node is the result of replacing the first goal with the body of a matching clause. If we traverse such a tree depth-first and from left to right, we get a series of lists of goals, resembling execution traces. This is just like serial Prolog execution.

Now suppose that we have a "pool" of *workers* [55]: modified WAMs or processes running on them. We begin by assigning a query (the root of the OR-tree) to one worker. It begins to work on the first clause that matches its first goal, i.e., the leftmost subtree. If other workers are idle, as of course they are initially, it assigns them the other subtrees, starting from the right. Each worker can do this with its own subtree, distributing subsubtrees of it to idle workers. If no other workers are idle, a worker simply executes its subtree by itself, just like an ordinary WAM, returning results to its parent. Then, when it can do no more, it becomes idle and waits for another task. Figure 7–18 gives the general idea: each number represents one of eight workers, and so the label of a node or tree tells which is working on it. Nodes **2** and **3** have been given the first available subtrees, **4** through **6** subsubtrees as they became available, and so forth.

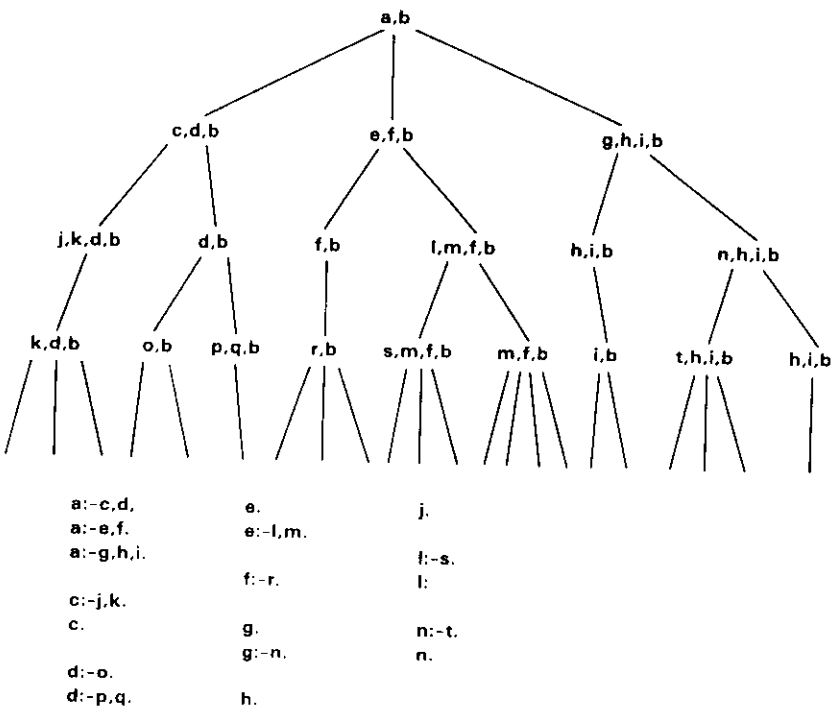


Figure 7–17 An OR-tree.

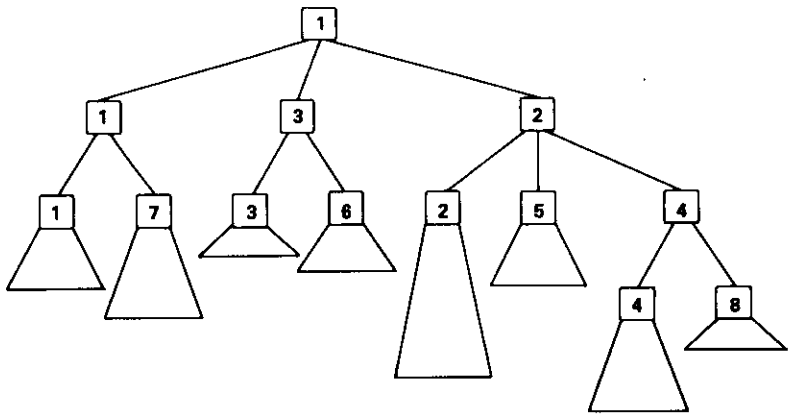


Figure 7–18 Dividing an OR-tree amongst workers.

The beauty of this scheme is that the available OR-parallelism, in theory, is always exactly right. We can give an idle worker its own subtree, or let a worker handle a large subtree on its own. If workers do not get tasks of roughly equal size, the ones that finish early can help the others. No worker is idle for long, and the workload seems to balance automatically. Compare this with the AOPM, which must restrain OR-parallel execution to prevent an exponential number of processes from being spawned, or with the RAP machine, which relies entirely on restricted AND-parallelism (perhaps too uncommon to be a mainstay [55]).

A major problem with implementing an OR-parallel machine is keeping track of the different bindings that different branches of execution make to the same variable [55]. Counting variations, about a dozen methods exist for doing this [56]. Most involve some sort of *binding array* or hash table for each worker or for each node of the OR-tree. For example, Warren's SRI model [55] gives each processor of a machine a local memory for storing a binding array.

Most of these machine models assume (or have been implemented on machines of) 16 to 32 processors, with very fast communication between them, e.g., with a shared global memory [12,44]. There are some exceptions, however [2]. The need to manage bindings quickly makes the tight coupling desirable. For instance, when a processor switches from one subtree to another, it needs to receive a set of bindings associated with the parent node of its new subtree. Whether these are readily available depends on the method for storing bindings; methods that save effort here require it elsewhere [56].

SUMMARY

Prolog is a reasonable alternative to other languages for implementing expert systems. It has more expressive power than conventional shells, yet shares many of their advantages, such as built-in inference and search mechanisms. As we have

seen, it has a simple, fairly tidy syntax and an elegant mechansim of execution. Furthermore, an execution engine, the Warren abstract machine (WAM), already exists for Prolog: the language translates directly into instructions for the WAM. The WAM can be emulated in software or microcode, or built; in any case, it provides much faster execution than an interpreter.

Parallel machines designed expressly for expert systems usually support production systems. Most architectures are conventional multiprocessors with some hardware for communications or managing memory; only one appears to be based on processors analogous to the WAM. The software that performs the production-system operations in parallel is much more important than the architectures themselves. Parallel Prolog machines can be conventional multiprocessors, but more often they are collections of Warren engines. They can exploit any of several principal types of parallelism.

Implementors of expert systems should seriously consider Prolog, for both its capabilities and its speed of execution. Those who prefer to use expert system languages such as OPS5 should realize that these need a standard, serial abstract machine similar to the WAM; without the WAM, Prolog compilers and Prolog machines would be far more primitive than they are at present. The execution engine of the PESA-I processing elements [42] might be a step in the right direction, but someone needs to make a careful study of the serial execution of production systems.

REFERENCES

1. *ALS Prolog Technical Reference Manual,* Syracuse, New York: Applied Logic Systems, 1987.
2. Ali, K. A. M., "OR-Parallel Execution of Prolog on a Multi-Sequential Machine." *International Journal of Parallel Programming* 15 (1987): 189–214.
3. Ben-David, A., and Sterling, L. "A Prototype Expert System for Credit Evaluation." In *Artificial Intelligence in Economics and Management,* edited by L. F. Pau. pp. 121–128. Amsterdam: Elsevier, 1986.
4. Carlsson, M., and Widen, J. *SICStus Prolog User's Manual, SICS Research Report R88007.* Swedish Institute of Computer Science, February 1988.
5. Clark, K. L., and McCabe, F. G. "PROLOG: A Language for Implementing Expert Systems." In *Machine Intelligence 10,* edited by Y.-H. Pao. Chichester, England: Ellis Horwood, 1982.
6. Clark, K. L., and Gregory, S. "Notes on the Implementation of PARLOG." *Journal of Logic Programming* 2 (1985): 17–42.
7. Clocksin, W. F., and Mellish, C. S. *Programming in Prolog.* Berlin: Springer-Verlag, 1984.
8. Conery, J. S. *Parallel Execution of Logic Programs.* Boston: Kluwer Academic Publishers, 1987.
9. Debray, S. K. *The SB-Prolog System, Version 2.2.1: A User Manual.* Tucson, AZ, University of Arizona, 1987.

10. DeGroot, D. "Restricted AND-Parallelism." *International Conference on Fifth Generation Computer Systems*. Tokyo, 1984, pp. 471–478.
11. DeGroot, D. "Restricted AND-Parallelism and Side-Effects." *1987 Symposium on Logic Programming*. San Francisco, September 1987, pp. 80–89.
12. Disz, T., Lusk, E., and Overbeek, R. "Experiments with OR-parallel Logic Programs." *Fourth International Conference on Logic Programming*. Melbourne, 1987, pp. 576–600.
13. Dobry, T. P., Despain, A. M., and Patt, Y. N. "Performance Studies of a Prolog Machine Architecture." *Twelfth Annual International Symposium on Computer Architecture*. Boston, 1985, pp. 180–190.
14. Dwork, C., Kanellakis, P. C., and Mitchell, J. C. "On the Sequential Nature of Unification." *Journal of Logic Programming* 1 (1984): 35–50.
15. Fagin, B. S., Patt, Y. N., Srini, V., and Despain, A. M. "Compiling Prolog into Microcode: A Case Study Using the NCR/32-000." *MICRO 18 Proceedings*. Pacific Grove, CA, 1985, pp. 79–88.
16. Fagin, B. S., and Despain, A. M. "Performance Studies of a Parallel Prolog Architecture." *Fourteenth Annual International Symposium on Computer Architecture*. Pittsburgh, 1987, pp. 108–116.
17. Gee, J., Melvin, S. W., and Patt, Y. N. "Advantages of Implementing Prolog by Microprogramming a Host General Purpose Computer." *Fourth International Conference on Logic Programming*. Melbourne, 1987, pp. 1–20.
18. Gupta, A., et al. "Parallel Algorithms and Architectures for Rule-based Systems." *Thirteenth Annual International Symposium on Computer Architecture*. Tokyo, 1986, pp. 28–37.
19. Hammond, P. "Micro-PROLOG for Expert Systems." In *Micro-PROLOG: Programming in Logic*, edited by F. G. McCabe. Englewood Cliffs, New Jersey: Prentice-Hall, 1984.
20. Hermenegildo, M. V. "An Abstract Machine for Restricted AND-parallel Execution of Logic Programs." *Third International Conference on Logic Programming*. London, 1986, pp. 25–39.
21. Hermenegildo, M. V., and Nasr, R. I. "Efficient Management of Backtracking in AND-parallelism." *Third International Conference on Logic Programming*. London, 1986, pp. 40–54.
22. Hermenegildo, M. V. "Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs." *Fourth International Conference on Logic Programming*. Melbourne, 1987, pp. 556–575.
23. Hogger, C. J. *Introduction to Logic Programming*. London: Academic Press, 1984.
24. Jackson, P. *Introduction to Expert Systems*. Workingham, England: Addison-Wesley, 1986.
25. Kaneda, Y., et al. "Sequential Prolog Machine PEK." *New Generation Computing* 4 (1986): 51–66.
26. Lloyd, J. W. *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1984.
27. Maier, D., and Warren, D. S. *Computing with Logic*. Menlo Park, CA: Benjamin/Cummings, 1988.
28. Maruyama, F., et al. "Prolog-based Expert System for Logic Design." *International Conference on Fifth Generation Computer Systems*. Tokyo, 1984, pp. 563–571.

29. Matsumura, Y., et al. "Consultation System for Diagnosis of Headache and Facial Pain: RHINOS." In *Logic Programming '85: Proceedings of the Fourth Conference,* edited by E. Wada, pp. 287–298. Berlin: Springer-Verlag, 1985.

30. Matwin, S., et al. "Logic-based Tools for Negotiation Support." *1987 Symposium on Logic Programming.* San Francisco, September 1987, pp. 499–506.

31. Miranker, D. P., "Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE." *International Conference on Fifth Generation Computer Systems.* Tokyo, 1984, pp. 449–457.

32. Miyazaki, J., et al. "MANJI: An Architecture for Production Systems." *Twentieth Annual Hawaii International Conference on System Sciences.* Honolulu, 1987.

33. Miyazaki, J., et al. "A Shared Memory Architecture for MANJI Production System Machine." In *Database Machines and Knowledge Base Machines.* edited by H. Tanaka, pp. 517–531. Boston: Kluwer Academic Publishers, 1988.

34. Naish, L. *Negation and Control in Prolog.* Berlin: Springer-Verlag, 1986.

35. Nakazaki, R., et al. "Design of a High-speed Prolog Machine." *Twelfth Annual International Symposium on Computer Architecture.* Boston, 1985, pp. 191–197.

36. Nitta, K., and Nagao, J. "KRIP: A Knowledge Representation System for Laws Related to Industrial Property." In *Logic Programming '85: Proceedings of the Fourth Conference,* edited by E. Wada, pp. 276–286. Berlin: Springer-Verlag, 1985.

37. Oshisanwo, A. O., and Dasiewicz, P. P. "A Parallel Model and Architecture for Production Systems." *International Conference on Parallel Processing.* St. Charles, IL, August 1987, pp. 147–153.

38. Pereira, F. C. N. *C-Prolog User's Manual.* Edinburgh: EdCAAd, 1983.

39. Poe, M. D. "Control of Heuristic Search in a PROLOG-based Microcode Synthesis Expert System." *International Conference on Fifth Generation Computer Systems.* Tokyo, 1984, pp. 589–595.

40. Robinson, J. "A Prolog Processor Based on a Pattern Matching Memory Device." *Third International Conference on Logic Programming.* London, 1986, pp. 172–179.

41. Sammut, R. A., and Sammut, C. A. "The Implementation of UNSW-Prolog." *Australian Computer Journal* 15 (1983): 58–64.

42. Schreiner, F., and Zimmerman, G. "PESA-I—A Parallel Architecture for Production Systems." *International Conference on Parallel Processing.* St. Charles, IL, August 1987, pp. 166–169.

43. Shapiro, E. Y. *A Subset of Concurrent Prolog and Its Interpreter, Technical Report TR-003.* Tokyo: Institute for New Generation Computer Technology, 1983.

44. Shen, K., and Warren, D. H. D. "A Simulation Study of the Argonne Model for Or-parallel Execution of Prolog." *1987 Symposium on Logic Programming.* San Francisco, September 1987, pp. 54–68.

45. Sterling, L., and Shapiro, E. Y. *The Art of Prolog.* Cambridge, MA: MIT Press, 1986.

46. Stolfo, S. J., Miranker, D. P., and Shaw, D. E. "Architecture and Applications of DADO: A Large Scale Parallel Computer for Artificial Intelligence." *Eighth International Joint Conference on Artificial Intelligence.* Karlsruhe, 1983, pp. 850–854.

47. Stolfo, S. J. "Initial Performance of the DADO2 Prototype." *Computer* 20 (1987): 75–83.

48. Taki, K., et al. "Hardware Design and Implementation of the Personal Sequential Inference Machine (PSI)." *International Conference on Fifth Generation Computer Systems.* Tokyo, 1984, pp. 398–409.

49. Tick, E. *An Overlapped Prolog Processor, Technical Note 308.* Menlo Park, CA: SRI International, 1983.

50. Tick, E., and Warren, D. H. D. "Towards a Pipelined Prolog Processor." *1984 Symposium on Logic Programming.* Atlantic City, February 1984, pp. 29–40.

51. Touati, H., and Despain, A. "An Empirical Study of the Warren Abstract Machine." *1987 Symposium on Logic Programming.* San Francisco, September 1987, pp. 114–124.

52. Ueda, K. "Guarded Horn Clauses." In *Logic Programming '85: Proceedings of the Fourth Conference,* edited by E. Wada, pp. 168–179. Berlin: Springer-Verlag, 1985.

53. Warren, D. H. D. *Applied Logic—Its Use and Implementation as a Programming Tool, Technical Note 290.* Menlo Park, CA: SRI International, 1983.

54. Warren, D. H. D. *An Abstract Prolog Instruction Set, Technical Note 309.* Menlo Park, CA: SRI International, 1983.

55. Warren, D. H. D. "The SRI Model for Or-parallel Execution of Prolog—Abstract Design and Implementation." *1987 Symposium on Logic Programming.* San Francisco, September 1987, pp. 92–102.

56. Warren, D. H. D. "Or-parallel Execution Models of Prolog." *1987 International Joint Conference on Theory and Practice of Software Development.* Pisa, Italy, March 1987, pp. 243–259.

57. Waterman, D. A., and Hayes-Roth, F. "An Investigation of Tools for Building Expert Systems." In *Building Expert Systems,* edited by D. B. Lenat, pp. 169–215. Reading, MA: Addison-Wesley, 1983.

58. Westphal, H., and Robert, P. "The PEPSys Model: Combining Backtracking, AND- and OR-parallelism." *1987 Symposium on Logic Programming.* San Francisco, September 1987, pp. 436–448.

59. Yoshino, H., et al. "Legal Expert System LES-2." In *Logic Programming '86: Proceedings of the Fifth Conference,* edited by E. Wada, pp. 34–45. Berlin: Springer-Verlag, 1986.