

INVESTIGATION OF  
A TAGGED COMPUTER ARCHITECTURE  
FOR THE PROLOG LANGUAGE

by

DANIEL R. MEIGS

B.S., Kansas State University, 1985

---

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY  
Manhattan, Kansas

1988

Approved by:

  
Major Professor

---

LD  
2668  
.R4  
EECE  
A88  
M44  
c.2

ALL207 312051

Table of Contents

List of Figures . . . . .	iv
List of Tables . . . . .	vii
Acknowledgments . . . . .	viii
1.0 Introduction . . . . .	1
2.0 Computer Architecture and Tags . . . . .	3
3.0 Prolog Overview and Interpreter Details . . . . .	7
3.1 Prolog Overview . . . . .	7
3.2 Interpreter Details . . . . .	10
4.0 Application of a Tagged Architecture to Prolog. . .	15
4.1 Data . . . . .	15
4.2 Control . . . . .	16
4.2.1 Decisions . . . . .	17
4.2.2 Tags . . . . .	18
Null Pointer . . . . .	18
Pointer . . . . .	19
Last Call . . . . .	20
EOS . . . . .	20
Code . . . . .	20
Complete Set . . . . .	20

4.3	Evaluation. . . . .	22
	Unification. . . . .	23
	Variable Binding . . . . .	26
5.0	Data Structures and Depth First Search. . . . .	28
5.1	Data Structures . . . . .	29
	Global Registers . . . . .	30
	Frame Stack and Trail Stack. . . . .	30
5.2	Algorithms. . . . .	35
	Initialization . . . . .	36
	Call Selection . . . . .	37
	Procedure Selection. . . . .	38
	Unification. . . . .	38
	Frame Creation . . . . .	39
	Backtracking . . . . .	39
	Unbinding. . . . .	39
5.3	Database Form . . . . .	40
6.0	Demonstration of Control Flow . . . . .	45
6.1	Likes: A Simple Example . . . . .	45
6.2	Sister-of: A More Comprehensive Example. . . . .	54
7.0	Conclusions. . . . .	80
	References . . . . .	81

### List of Figures

2.1	Data format of a 32-bit word with an 8-bit tag in a tagged machine. . . . .	5
3.1	Example database. . . . .	12
4.1	The tree structure of parameters of a call. . . . .	24
4.2	Address generation concept for unification routines . . . . .	26
5.1	Frame stack . . . . .	32
5.2	Backtrack example . . . . .	34
5.3	An example Prolog program . . . . .	40
5.4	Database structure. . . . .	41
5.5	Codified facsimile of an example program. . . . .	42
5.6	Sisterhood example. . . . .	43
5.7	Codified facsimile of the sisterhood example program. . . . .	44
6.1	Frame stack after initialization. . . . .	47
6.2	Global registers after initialization . . . . .	47
6.3	Global registers after creation of second frame . . . . .	49
6.4	Trail stack after creation of second frame. . . . .	50
6.5	Frame stack after creation of second frame. . . . .	50
6.6	Global registers at unification . . . . .	51

6.7	Trail stack after creation of third frame . . . . .	52
6.8	Frame stack after creation of third frame . . . . .	53
6.9	Frame stack after initialization. . . . .	56
6.10	Global registers after initialization. . . . .	56
6.11	Global registers after frame 2 . . . . .	58
6.12	Frame stack after creation of second frame . . . . .	59
6.13	Trail stack after frame 2. . . . .	59
6.14	Global registers at unification. . . . .	60
6.15	Global registers after frame 3 . . . . .	61
6.16	Frame stack after creation of the third frame. . . . .	62
6.17	Global registers after frame 4 . . . . .	63
6.18	Trail stack after frame 4. . . . .	63
6.19	Frame stack after creation of fourth frame . . . . .	64
6.20	Frame stack after creation of the fifth frame. . . . .	67
6.21	Global registers after frame 5 . . . . .	67
6.22	Trail stack after frame 5. . . . .	67
6.23	Frame stack after creation of the sixth frame. . . . .	69
6.24	Global registers after frame 6 . . . . .	70
6.25	Trail stack after frame 6. . . . .	70
6.26	Global registers after backtracking. . . . .	73
6.27	Frame stack after creation of the sixth frame. . . . .	75
6.28	Global registers after frame 6 . . . . .	75
6.29	Trail stack after frame 6. . . . .	75
6.30	Global registers after frame 6 . . . . .	76
6.31	Trail stack after frame 6. . . . .	76
6.32	Frame stack after creation of the sixth frame. . . . .	78

6.33 Global registers after last solution . . . . .	79
---	----

List of Tables

2.1	A proposed set of tags in a general purpose computer . . . . .	6
4.1	Eighteen tags identifying the Prolog data . . . . .	16
4.2	Control procedures and decisions. . . . .	18
4.3	Decisions based on the null pointer tag . . . . .	19
4.4	Tags for Prolog . . . . .	21
4.5	Parameter matching rules. . . . .	25
5.1	The set of global registers . . . . .	31

### Acknowledgments

Not even the humblest project can be accomplished in a vacuum, and I owe a debt of gratitude to so many people who helped with this project that I can not name them all. However, I would like to express my gratitude to Dr. E. Haft for the guidance and help he gave me on this project and, most of all, for his patience while I was incommunicado. I would also like to thank Dr. E. Unger and Dr. E. Fowler for serving on my graduate committee.

Special thanks go to Dr. Fowler for his support and "fatherly advice" (life would have been easier if I could have followed it more closely), to Denise Bandat and NeoGraphics of Phoenix, Arizona for excellent printing services, and to a half a dozen close friends who lent support and encouragement without which this report may not have been completed. I must also include a tip of the hat to my parents, whom I can not thank enough.



## 1.0 Introduction

During the past 40 years the nature of computer programs has changed radically. Today's artificial intelligence research is dealing with problems of enormous complexity. Computers are being used for vision systems, natural language understanding, robotic control, and attempts to imitate human reasoning. These tasks are far more sophisticated than the primarily numerical processing of the early days of computing.

Sophisticated tasks require sophisticated tools. As computer programming problems have evolved, computer programming languages have become more sophisticated as well. Prolog, the primary artificial intelligence language of Europe and Japan, is a language based on first order logic. Unlike most programming languages which require the programmer to tell the computer what to do and when to do it, Prolog programs tell the computer facts about the world and ask it to draw conclusions [3:253].

While programming languages have become more sophisticated, the computers on which the languages have

been run have not. According to Elaine Rich [5:405], "The idea of designing an A.I. machine has been around since at least 1960..." However most of the research has been concerned with the list processing operations and the memory management necessary for implementing the LISP language. Only in the past decade has serious attention been paid to a machine based on Prolog.

The goal of this paper is to provide an explanation of how the inference mechanism of a Prolog interpreter works and to suggest a tagged computer architecture which will facilitate the implementation of a Prolog interpreter. The organization of the report is as follows. Chapter 2 introduces the general idea of a tagged computer architecture. Chapter 3 introduces the Prolog language and the theory of first order logic. Chapter 4 relates the concept of tags to Prolog and suggests a set of tags that can help optimize a Prolog computer. Chapter 5 describes in detail the form of a Prolog database and the data structures that are used to record the state of the solution. Chapter 6 details the execution of two Prolog programs showing how all of the elements of chapter 5 are used and how the tags of chapter 4 play a part in the process. Chapter 7 suggests an area of further research.

## 2.0. Computer Architecture and Tags

There are many architectural issues in the design of a computer. The most fundamental issue is the nature of the data in the machine. Most common computers use a single, linear memory with a fixed number of bits allocated for every word. In this system, called the von Neumann architecture, a specific binary word has no intrinsic meaning out of the context of the words around it and no distinction is made between data and programs. Meaning is attached to the data words when they are manipulated by the computer or program in which they exist.

There are many disadvantages to the von Neumann architecture. The biggest disadvantage is that there is a great conceptual distance between the problem that a computer program is trying to solve and the hardware on which the program is being run. This conceptual distance is known as the semantic gap [1]. To narrow the semantic gap, it makes sense to tailor the hardware, operating system, and software to the task.

There are several ways to tailor the system to the problem. The most flexible approach, and the easiest to implement, is to use self-identifying data throughout the machine. That is, each binary word, whether it is code or data, should have meaning out of the context of the program in which it is written. The hardware in such a machine could infer much of the necessary manipulation based on the type of data that was being manipulated.

For example, if the machine could tell the difference between integers, real numbers, boolean expressions, complex numbers, etc., the machine language would need only one ADD instruction. The advantage would be two-fold. First, hardware could "know" to behave differently if the two numbers were integers than if the two numbers were complex. Second, the machine could have built in traps for error handling. If the programmer tried to add something that was not a number, such as an instruction or a boolean, or if the programmer tried to manipulate an uninitialized variable the machine would know to execute some error handling routines. This could provide some powerful diagnostics.

The way to make data self-identifying is to use bit-fields called tags. Using tags, a data item could have two bit-fields as shown in figure 2.1. One field (the value field) would contain the binary data just as it would in a von Neumann machine. The other field (the tag field) would contain a set of bits that identified the data as an

instruction, an address, a real number, etc. Edward Feustel [2] identified 32 types of data that should be represented in a general purpose computer. Mr. Feustel's suggestions included tags for such primitives as integers and real numbers, as well as tags for common data structures such as single and double linked lists; stacks and queues; matrices, vectors, and sparse vectors. Mr. Feustel also recommended tags for system information such as machine states, messages, interrupts, and garbage. The full set of tags is listed in table 2.1.

---

TAG 8-bits	VALUE 24-bits
---------------	------------------

Figure 2.1. Data format of a 32-bit word with an 8-bit tag in a tagged machine.

---

It may be possible to go beyond Mr. Feustel's tags for a general purpose machine and develop a specialized set of tags for a specific machine dedicated to a specific language. Such specialization has the potential to improve greatly the efficiency of the execution of the language. However this requires an understanding of the language to be implemented. Chapter 3 provides an overview of the Prolog language and the theory of first order logic behind Prolog.

---

Table 2.1. A proposed set of tags in a general purpose computer. Source: Feustel [1].

integer  
real number  
double precision integer  
double precision real  
single precision complex  
double precision complex  
undefined  
mixed types  
character  
Boolean  
vector of  
reference to  
label in ith environment  
matrix of  
sparse vector of  
single linked list of  
double linked list of  
stack of  
queue of  
machine state of  
message from-to  
interrupt of  
event  
parameter set for  
procedure-environment designator  
name of variable  
i.d. of process or user  
instructions  
file  
formal parameter  
semaphore  
garbage

---

### 3.0 Prolog Overview and Interpreter Details

The previous chapter described how a tagged architecture could improve the efficiency of a general purpose computer. A set of tags was outlined based on a paper by E. Feustel. The purpose of this report is to describe an efficient computer architecture for a computer dedicated to the Prolog language. This chapter attempts to explain the nature of Prolog programs and how a Prolog interpreter executes the programs. An understanding of this information is essential to the design of a set of tags for a Prolog computer. It is assumed that the reader has some knowledge of Prolog.

#### 3.1 Prolog Overview

A Prolog program consists of a database of facts and rules describing the problem that the program is designed to solve. A user asks a query and the interpreter answers the query based on the facts and rules in the database. The database for a simple problem concerning relationships between siblings is explained below. The database contains one rule and three facts.

The rule in the database could state that one person is the sister of a second person if the first person is a female and the two people have the same parents. In Prolog the rule would be written with a head and a body separated by the symbol ":-" which is read as "if." Prolog rules are terminated by a period. The rule then would be written as follows.

```
sister-of (A,B) :- female(A), same-parents (A,B).
```

The head of the rule is `sister-of (A,B)`. The head of the rule describes what fact the rule intends to define. The body, in this case `female (A), same-parents (A,B)`, describes the conjunction of goals that must be satisfied, one after the other, for the head to be true. The comma in the body is a conjunction and is read, "and." In Prolog rules, the scope of a variable is the entire rule from the head to the period. So, if the variable `A` is instantiated to `alice`, the interpreter will try to satisfy the goals `female (alice)` and `same-parents (alice,B)`. (Note that in Prolog, variables begin with upper case letters and atoms begin with lower case letters.)

The first two facts included in the database could state that Mary and Sue are females. These facts would be written in Prolog as follows.

```
female (mary).
```

```
female (sue).
```



The third fact could state that Sue and John have the same parents. In Prolog, that fact would be written as follows.

```
same-parents (sue, john).
```

After the rule and the facts have been entered into the database, the program would contain the following four lines:

```
sister-of (A,B) :- female(A), same-parents (A,B).
female (mary).
female (sue).
same-parents (sue, john).
```

The user, then, could ask the query, "is Sue the sister of John?" The query would be written as follows.

```
:- sister-of (sue, john).
```

The Prolog interpreter would consult the database and respond that Sue was indeed John's sister.

There are three important things about Prolog programs that are illustrated by this example. First, note that the fact that Mary is a female is of no use in answering the query, "is Sue the sister of John?" Not all facts in a database are applicable to all questions. Second, if the user asked, "Is Mary the sister of John?" the interpreter would answer "No." Although it may be true in some family that Mary is the sister of John, that relationship cannot be proven from the database. Third, while the database shows that Sue is the sister of John, it does not prove that John is the brother of Sue. The query, "Is John the brother of

Sue?" would fail. When the interpreter answers no to a question it means not provable; it does not mean not true. [3]

The previous example simply asked if Sue is John's sister. If that was all that one could do with the database, Prolog would not be very useful. One might wish to inquire about more general cases such as "Does John have a sister?" "Who is John's sister?" "Does anyone have a sister named Sue?" "Who has a sister named Sue?" and, finally, "Who is the sister of whom?" Fortunately, Prolog allows variables in queries and returns the names of any atoms that, when put in place of the variables in the query make the query true. Thus, if one really did care to know the name of John's sister, one could ask the following query.

```
:- sister-of (X, john).
```

The response that the interpreter would give is as follows.

```
X = sue
```

The response means that Sue is the only member of the database that makes the statement `sister-of (X, john)` true.

### 3.2 Interpreter Details

Formally, the rule:

```
sister-of (A,B) :- female(A), same-parents(A,B).
```

is a first order logic clause. The clause can be read, "For all values of A and B, it is true that A is the sister of B if and only if it is true that A is a female and it is true that A and B have the same parents." Because the rule is

stated as an if and only if relationship, a second interpretation of the rule is valid: "It is not true that A is the sister of B if it is not true that A is a female or if it is not true that A and B have the same parents." Furthermore something is assumed not true if it cannot be proven true. Therefore, the preceding interpretation can be stated more generally as follows. The head of a rule is proven not true as soon as one of the goals in the body is not proven true. This interpretation is the most important because it is the way the Prolog interpreter uses the rule in answering a query.

The strategy that the Prolog interpreter uses to answer a query is known as resolution. The technique is to negate the query and then see if the negation is inconsistent with the database. In other words, the interpreter assumes that the answer to the query is, "No." Then the interpreter matches the query with facts and rules in the database. If a fact matches the query, then the assumption that the answer was no is shown to be inconsistent with the database. Under those conditions, the interpreter answers, "Yes." If the query matches the head of a rule, the goals in the body of the rule become new queries, and the process is repeated recursively. If all of the rules and facts in the database can be searched without disproving the initial assumption that the query is false, the interpreter will answer, "No."

In the above example, the query was

```
:- sister-of (sue, john).
```

The interpreter, in trying to answer the query would assume that Sue is not the sister of John. This assumption will be regarded as valid unless something in the database contradicts it. The interpreter would match the query with the head of the rule that defines the sister-of relationship:

```
sister-of (A,B) :- female(A), same-parents (A,B).
```

Then the interpreter would substitute the atom `sue` for all occurrences of the variable `A`, and it would substitute the atom `john` for all occurrences of the variable `B`. This substitution would leave the interpreter with the two goals of the body:

```
female(sue)
same-parents (sue, john)
```

The state of the database is shown in figure 3.1.

---

Database:

```
R1: sister-of(A,B) :- female(A), same-parents(A,B).
```

```
F1: female(mary).
```

```
F2: female(sue).
```

```
F3: same-parents(sue, john).
```

```
Query: :- sister-of(sue, john).
```

```
Second goal: female(sue).
```

```
Third goal: same-parents(sue, john).
```

Figure 3.1 Example database

---

The query, :- sister-of (sue, john). matches with the rule, R1. The result of matching is that A is instantiated to John. Because the query matched the head of a rule, there are more goals to satisfy corresponding to the body of the rule. The interpreter, then, begins again with the second goal, female(sue). If nothing in the database shows that Sue is a female, then the head of the rule is proven false and the second goal (same-parents (sue, john)) is not needed. Recall that a rule is assumed false unless all parts of the rule are proven true. However, in this database, there is a fact that states that Sue is indeed a female. Consequently the interpreter turns its attention to the same-parents goal.

The interpreter follows the same strategy of assuming that Sue and John do not have the same parents. If the database fails to show that Sue and John have the same parents, then the head of the rule is shown to be false and cannot contradict the assumption that Sue is not the sister of John. In that case, the interpreter would search the rest of the data base for rules or facts beginning with sister-of. In this example, however, the goal same-parents (sue, john) is proven true by the third fact in the database:

same-parents (sue, john).

If that fact were replaced by a rule governing the notion of same-parents, then the interpreter would have other goals to consider in resolving the goal same-parents(sue, john).

Thus the interpreter goes from the query through the database searching for facts and rules that contradict the assumption that the query is false. When the query is further defined by a rule, the clauses of the rule make up new goals to resolve. If the entire database can be searched without proving the goal true, then the interpreter has proven the goal false.

## 4.0 Application of a Tagged Architecture to Prolog

This chapter relates the concept of tags, as discussed in Chapter 2, to Prolog and suggests a set of tags that can help optimize a Prolog computer.

### 4.1 Data

Prolog has a different set of data than one would find in a general purpose programming language. There are three data structures in Prolog: constants (integers and atoms), structures (lists), and variables. The code consists of assertions (facts), procedures (rules), and built in predicates (fundamental rules). Variables in Prolog can be either bound or unbound. If a variable is bound, it can be bound to an integer, structure, or another variable. Furthermore, some Prolog systems make provisions for all of those variables to be either local, global, or void. This set of data types dictates a set of eighteen tags. These tags are listed in table 4.1, below.

---

Table 4.1. Eighteen tags identifying the Prolog data.

Constants:  
    integer  
    atom

Structure:  
    list

Local Variables  
    unbound  
    bound to integer  
    bound to atom  
    bound to structure  
    bound to variable

Global Variables  
    unbound  
    bound to integer  
    bound to atom  
    bound to structure  
    bound to variable

Void Variables  
    unbound  
    bound to integer  
    bound to atom  
    bound to structure  
    bound to variable

Code:  
    assertion  
    procedure  
    built in predicate

---

## 4.2 Control

The computer dealt with in this report has an architecture designed to run the Prolog language efficiently. This design required that the Prolog interpreter, the program that executed the user's code, be implemented efficiently. The specific control information



needed to implement a Prolog interpreter was determined by examining the control algorithm such an interpreter would use. The goal of the design was to make most of the decisions that need to be made during control flow simply by examining the tags of one or two data items. The control algorithm will be described in detail in chapter 5 below. At this point, the decisions that need to be made and the tags that facilitate them will be introduced.

#### 4.2.1 Decisions

The control algorithm outlined in David Rodenbaugh's master's report [4] included four main procedures: call selection, procedure selection, frame creation, and backtracking. Each of these procedures required decisions. The four procedures and their decisions are listed in table 4.2.

The set of tags in table 4.1 is sufficient to make two of the decisions. The first decision during call selection, "Is the current procedure an assertion?" can be made by comparing the tag of the current procedure register to the tag of an assertion. Likewise, the fourth decision during call selection, "Was a call found?" can be made by comparing the tag of the current call register to the tag of a procedure. The rest of the decisions require specific control information.

---

Table 4.2 Control procedures and decisions.

Call Selection

- 1) Is the current procedure an assertion?
- 2) Is the current call pointer null?
- 3) Is the most recent parent not frame 1?
- 4) Was a call found?
- 5) Are there any candidates left?

Procedure Selection

- 1) Is the current procedure null?
- 2) Is the next candidate null?
- 3) Did the current procedure fail to unify with current call?

Frame Creation

- 1) Is the current call the last call of the procedure?
- 2) Are there other candidate procedures?

Backtracking

- 1) Is there a backtrack point?
  - 2) Has anything been put on the trail during this frame?
- 

#### 4.2.2 Tags

##### Null Pointer

A null pointer tag would be the single most helpful piece of control information. Of the 10 remaining decisions, six ask if one of the pointers is null or if there are any calls or candidates left. Those calls are listed in table 4.3 below.

If the last candidate in the linked list of candidates was linked to a word with the tag "null pointer," then the decisions "Is the next candidate null?" "Are there any other

---

Table 4.3. Decisions based on the null pointer tag listed by function.

Next Candidate:

- Are there any candidates left?
- Is the next candidate null?
- Are there other candidate procedures?

Current Call:

- Is the current call pointer null?

Current Procedure:

- Is the current procedure null?

Most Recent Backtrack:

- Is there a backtrack point?
- 

candidate procedures?" and "Are there any candidates left?" could be made by comparing the tag of the next candidate pointer to the null pointer tag.

Pointer

The second tag that would be useful is the pointer tag. E. Feustel suggested the pointer tag in a general machine. Most of the work that is handled by the interpreter deals with manipulating pointers. The interpreter could check that all of pointers that it manipulates really are pointers. This could provide some internal error handling.

### Last Call

One of the decisions during Frame Creation is "Is the Current Call the last call of the procedure?" If the interpreter could affix a "Last Call" tag to the last call of each procedure when the database is first organized, this decision could be taken care of automatically. Most calls would have a "Procedure" tag. The last call of each procedure would have a "Last Call" tag. If the two tags differed only by the last bit, then the distinction could be overlooked easily when necessary.

### EOS

The list of tags in table 4.1 includes a tag for structures. Generally, lists are easier to handle if they are terminated by a special symbol. The "EOS" tag is suggested for this purpose.

### Code

The machine code that makes up the interpreter, operating system, etc. should be distinguished from the rest of the words in memory. This is important to prevent the user of the machine from corrupting the important code. The "Code" tag is suggested for this purpose.

### Complete Set

The set of tags suggested in this chapter appear along with the tags of Section 4.1 in table 4.4 below. The actual

binary value of the tags are, for the most part, hypothetical. However there are some important points.

---

Table 4.4 Tags for Prolog

<u>Type</u>	<u>Tag (8 bits)</u>	<u>Value (24bits)</u>
Local Variables	1000xxxx	
Global Variables	1010xxxx	
Void Variables	1100xxxx	
unbound	1xxx1011	variable index
bound to integer	1xxx1100	variable index
bound to atom	1xxx1101	variable index
bound to structure	1xxx1110	variable index
bound to variable	1xxx1111	variable index
Integer	00000000	immediate data
Atom	00000001	pointer to heap
List	00000010	pointer to beginning
EOS	00000011	(don't care)
Last Call	00000100	pointer to skeleton
Procedure	00000101	pointer to skeleton
Assertion	00000110	pointer to skeleton
Code	00000111	machine code
Pointer	00001000	address
Null Pointer	00001001	(don't care)
Built in Predicate	00001010	address of procedure

---

First, variables can be readily distinguished from the other tags because only variables have a one as the most significant bit. Second, the last four bits of a variable are determined by what, if anything, it is bound to. This was done because the unification routine makes a decision based on what a variable is bound to, and all variables bound to the same data type must look alike. Third, the three types of variables: local, global, and void notwithstanding, there are sixteen tags. Each tag has a unique value in the four least significant bits. Thus for most applications only a four bit tag need be manipulated. Fourth, often the differences between very similar data structures represented with unique tags will be disregarded. "Last Call" and "Procedure" both identify calls. "Pointer" and "Null Pointer" are both valid pointers. "Integers" are just a special case of "Atoms." To allow the differences between these tags to be ignored easily when appropriate, the pairs of tags differ only in the least significant bit.

#### 4.3 Evaluation

The goal of adding the tags was to answer automatically most of the decisions necessary during control flow. There were twelve decisions listed in table 4.2. Two of the decisions were answered based on a data type tag. Six of the decisions were made based on the null pointer tag. One of the decisions was made based on the last call tag.

Of the three remaining decisions, two of them cannot be made more easily with tags. The decision, "Is the Most Recent Parent not frame 1?" can best be answered by checking the value of the Most Recent Parent pointer. The decision, "Has anything been put on the trail during this frame?" can best be answered by comparing the Top of Trail register to the Trail Pointer stored in the frame.

#### Unification

The final decision, "Did the Current Procedure fail to unify with the Current Call?" is the most complicated of all. This decision invokes the Unification procedure. Unification is the process of comparing the parameters of the Current Call to the parameters of the head of a candidate procedure to see if they match.

Because the parameters of a call can be atoms, variables, or structures, the parameters of a call can be conceptualized as a tree. The tree structure is illustrated in figure 4.1 below. The call name is the root of the tree, and the parameters are siblings on the first level of nodes. If a node of the tree is a structure, the parameters of that structure are descendants of the node. In this manner, the tree is built recursively.

The unification procedure traverses the tree for the call and the tree for the procedure head trying to match corresponding parameters. If the two parameters are atoms, then they match if they both point to the same atom in the

---

Call: **a (B, c (X,y), d)**

Tree:

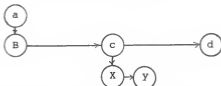


Figure 4.1 The tree structure of parameters of a call.

---

heap. Thus if the two tags are atoms, the value fields must simply be equal. Unification is more complicated for variables. Unbound variables match with anything, atoms, structures, or other variables. A variable that is bound to something matches with a parameter if the item to which the variable is bound matches the parameter.

There are nine ways that the three data types can appear in the unification attempt. The rules for parameter matching are listed in table 4.5.

The computer proposed in this paper would contain a set of six routines in memory corresponding to the six sections in table 4.5. The four least significant bits of both tags of the parameters that were being matched could be concatenated to form an eight bit value. This value could be a pointer to a segment of memory which contained a vector table. The vector table would contain the addresses of the six routines. Then the decision of which routine to jump to



Table 4.5 Parameter matching rules. Source: Rodenbaugh [4:44].

<u>Parameter</u>	<u>Parameter</u>	<u>Rule</u>
atom	atom	Matches if pointers are equal.
atom	variable	Matches if variable is unbound or bound to atom with same pointer.
variable	atom	
atom	structure	Never matches because atom has no children.
structure	atom	
variable	structure	Matches if variable is unbound or if variable is bound to a structure whose parameters match according to the rules in this table.
structure	variable	
structure	structure	Matches if all parameters match according to the rules in this table.
variable	variable	Matches if at least one variable is unbound. Matches if both are bound and the values they are bound to match according to the rules in this table.

for the current step in the unification procedure would be made automatically. This concept is illustrated in figure 4.2 below.

---

Parameter One:

Tag1 xxxx0000	Value1
------------------	--------

Parameter Two:

Tag2 xxxx1101	Value2
------------------	--------

Vector Table Offset:

Tag1 | Tag2  
0000 | 1101 = \$0D

Vector Table    Offset

	\$FF
	<----- \$0D    Address of routine for an atom \$00                and a variable bound to an atom

Figure 4.2. Address generation concept for unification routines.

---

### Variable Binding

During resolution, variables that are encountered are kept on a stack. The variables in a procedure head in the database are initially unbound. Thus all variables in the database will have the unbound variable tag (1xxx1011). The value field of an unbound variable contains the index of

that variable, a unique value for all unique variables in the database. When the variables are matched to the parameters of the call, they become bound and need to acquire a new tag and some additional information.

The new tag indicates to what data structure the variable is bound as outlined in table 4.4. The additional information includes a value pointer and, optionally, an environment pointer. The value pointer points to the specific data item to which the variable is bound. Thus if the variable is bound to an atom, the tag would reflect that. The value field would contain the variable index, and the next field, the value pointer, would contain a pointer to the specific atom in the atom heap. If the variable is bound to a structure, the environment pointer is necessary. The environment pointer is a pointer to a location in the stack where the value of the structure's parameters are defined.

## 5.0 Data Structures and Depth First Search

As the interpreter goes through a Prolog program trying to resolve a query, there is often a multitude of paths through the database. The search through the database begins with the initial query. This query becomes the current call. The interpreter then tries to find a fact or rule that matches the current call by name. The set of all facts and rules that match the current call are possible candidates for resolving the query. To complete the search, each of these paths must be searched. Prolog's search strategy is depth first. Each path is searched to its conclusion, either a successful conclusion or a dead end, before alternate paths are searched.

Depth first search is analogous to solving a maze by starting off in one direction and proceeding to either a solution or a dead end. If the path hits a dead end, there are many untried paths between the dead end and the start. The maze solver then backs up to the closest untried path and starts off again. If there is a path to the end of the maze, that strategy will find it. The problem with solving

a maze depth first is that there is a lot of record keeping necessary to prevent getting lost. A careful log of untried paths must be kept, and any information picked up along the way to a dead end must be discarded when a new path is taken. Therefore, the maze solver must keep track of where information was picked up.

Just as the hypothetical maze solver has to keep careful records, so must the computer. The interpreter can find a series of goals that succeed and instantiate several variables on the way to a dead end. At that point the interpreter must backtrack and try another solution path. Backtracking involves unbinding any variables that were instantiated on the way to the dead end and resetting the database pointers to the state they were in before the path that lead to the dead end was undertaken. The interpreter keeps the records needed to recover from a dead end in three data structures: a frame stack, a trail stack and a set of pointers. This chapter explains the form of the database, and these data structures. Chapter six explains how the interpreter uses the database and data structures and how tagged data can play a part.

### 5.1 Data Structures

The interpreter keeps a set of data structures to define the state of the search through the database. This information includes the current variable bindings, the information needed to find the next call, and the

information needed to recover from a dead end. The data structures used are the frame stack, the trail stack and a set of eight global registers.

#### Global Registers

The state of the search is kept in a set of global pointer registers. There is a total of eight registers in the set. The set of global registers is shown in table 5.1 below. The set of registers can be divided into three groups. The first two registers, Current Call, and Current Procedure define the position of the search through the database. The next three registers, Next Candidate, Most Recent Parent and Most Recent Backtrack define the next call if the Current Call and the Current Procedure don't match. The last three registers Top of Frame Stack, Last Top of Frame Stack, and Top of Trail are stack pointers used to keep track of the current positions on the stacks.

#### Frame Stack and Trail Stack

The frame stack is an area of memory in which information about variable bindings and the location of the next call is stored. This information changes as solution paths are searched and rejected. For this reason, the information is kept in a stack of frames with each frame representing the variable and call information at a point in the solution. Each time a call matches with the head of a rule, a frame is created. The frame contains five pointers,

---

Table 5.1 The set of global registers

Current Call  
Current Procedure  
Next Candidate  
Most Recent Parent  
Most Recent Backtrack  
Top of Frame Stack  
Last Top of Frame Stack  
Top of Trail

---

which determine the next call to be solved, an unknown number of variables with their bindings, and one pointer for record keeping within the stack. Figure 5.1 shows the generic form of the frame stack.

The space at the top of a frame contains the information about all the variables in the rule for which the frame was created. The number of variables that will be encountered varies from frame to frame. This provides the motivation for the previous frame pointer. The previous frame pointer is the first item in the frame and contains the address of the first entry in the previous frame. The other members of the frame are pointers that determine which call should be solved next and help recover from dead ends.

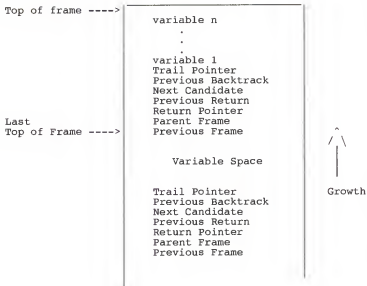


Figure 5.1 Frame stack.

In addition to the previous frame pointer, each frame has, as a minimum, a return pointer and a parent frame pointer. The return pointer points to the next call to be solved. The return pointer normally points to the call following the current call. If the current call is the last call in the rule, the return pointer is set to null. If the interpreter finds a null return pointer while looking for the next call, the interpreter must examine the return pointer of the frame that called the current frame. That



frame is called the parent frame. The parent frame pointer is the pointer used to trace back to a non-null return pointer.

In addition to the parent frame pointer and the return pointer, some frames have four other pointers. Sometimes in searching the database the interpreter will reach a call that fails to match with anything in the database, that is, it cannot be proven true by the database. Often when such a dead end is reached, there are untried paths between the dead end and the beginning. This is the case when there were untried candidate procedures that might have succeeded had they been tried. In this case the interpreter must backtrack to the point where an untried path exists and begin again. All variables that were instantiated due to calls beyond the backtrack point need to be unbound because their bindings are not valid. The trail stack and the four remaining pointers in the frame stack are used to allow backtracking. Figure 5.2 gives an example database which contains a backtrack point.

When the interpreter reaches a point where there is more than one possible candidate to match a call, the first of the candidates is chosen and a frame is created. If this candidate fails, the interpreter must be able to disregard any variable bindings made since that frame was created. The trail stack is used to record these potentially invalid bindings.

---

```
Database:
a :- b, c.
b :- d.
b :- e.
e.
c.
:- a.
```

Figure 5.2 Backtrack example. In this database the rule `b :- d` matches the call `b`. At that point the rule `b :- e` is an untried candidate for the call `b`, so the frame created for `b :- d` is a backtrack frame. When the call `d` fails the interpreter will backtrack and try the rule `b :- e`. The call `e` will succeed and the next call will be `c`.

---

As bindings are made in a backtrack frame, the interpreter puts a pointer on the trail stack which points to that variable's position in the frame. Then if backtracking is necessary, the bindings pointed to by all of the pointers between the top of the trail stack and the top of the trail stack when the frame was created need to be unbound. To do this, the interpreter must record the position of the top of trail at the time of creation of the backtrack frame. That position is recorded in the trail pointer location of the frame.

The other three pointers in the frame are very straight forward. A backtrack frame is created because there is at least one untried candidate that matches with the current call. This candidate is remembered by the next candidate pointer in the frame. The call that invoked the backtrack frame must be remembered to restore the state of the search upon backtracking. That call is pointed to by the previous return pointer. If there was a backtrack frame before the current one, a pointer to it must be kept in case further backtracking is necessary. That pointer is kept in the previous backtrack cell of the frame.

## 5.2 Algorithms

The following pseudo-code outlines the inference mechanism of the interpreter. The code is a data flow description of the machine proposed in this report. The algorithm is an adaptation of the algorithm by Rodenbaugh [4]. These algorithms will be used in the description of the execution of example programs in Chapter 6.

## Initialization

```
NC := null ptr.           {next candidate}
MRB := null ptr.         {Most recent backtrack}
MRP := null ptr.         {most recent parent}
TOT := Trail bottom      {top of trail}
LTOF := frame stack bottom {Last top of frame}
TOF := frame stack bottom {top of frame}
M(TOF) := null ptr       {previous frame pointer}
TOF := TOF + 1
M(TOF) := null ptr       {parent frame pointer}
TOF := TOF + 1
M(TOF) := null ptr.     {return pointer}
TOF := TOF + 1
CP := location of codified {current procedure}
      facsimile of main goal.
CP := CP + 1             {current procedure pts to
                        variable count for
                        the proc.}
VC := M(CP)             {variable count}
While VC > 0 do
  find the variable
  M(TOF) := var
  TOF := TOF + 3
  VC := VC - 1
go to call selection

{push the var on
the frame}
{leave room for environ.
and value pointer fields}
```

## Call Selection

```

temp := CP                                {temporary register}
If tag [M(CP)] = assertion                {is the current procedure
                                         an assertion}
then
  CC := M(LTOF + 2)
  while CC is null ptr. and MRP <> 0
  do
    CC := M(MRP + 2)
    MRP := M(MRP + 1)    {MRP := parent of MRP}
  if CC is null ptr.
  then
    output solution
    go to backtrack step
else
  While tag[M(temp)] <> procedure or last call {CP is a rule}
  do                                           {find a call}
    temp := temp + 1
    CC := temp                                {pointer to the call in CC}
    MRP := LTOF                               {current frame is a parent}

    {At this point current call points
     to the linked list of candidates.
     The procedure pointer of the list
     points to the first entry of the
     first candidate in the facsimile
     area. That is the next candidate
     pointer}

  if M(CC).procptr <> nil                    {if there is a candidate}
  then
    NC := M(CC).procptr                    {CC pts to the next
                                           candidate}

    go to procedure selection
else
  go to backtrack

```

### Procedure Selection

```
M(TOF + 5) := TOT {save top of trail in trail ptr}
M(TOF + 2) := CC  {save current call in return ptr}
CP := NC

if NC <> null ptr.
then
  NC := M(NC)
  CP := CP + 1 {current procedure pts to variable
               count}
  VC := M(CP)

while CP is not null ptr.
do

  go to unify procedure

  if CP did not unify with CC
  then
    CP := NC + 1
    if M(NC) is not null ptr.
    then
      NC := M(NC)           {get next candidate}
      VC := M(CP)
      unbind variables bound during
      unsuccessful unification attempt
    else
      go to frame creation  {CC unified with CP}

  go to backtrack          {CC did not unify with any
                           candidates}
```

### Unification

```
CP := CP + 1           {find first parameter
CC := CC + 1           of each}
Repeat until done
  temp := M(CP)        {generate the
MAR := address gen (temp, M(CC)) address of the
IR := M(MAR)          match routine
                       and jump there}

  if there is a match
  then
    do variable bindings
    find next parameter
  else return (failure)

restore CC
CP := CP + 1
Return (success)
```

### Frame Creation

```
M(TOF) := LTOF           {push previous frame pointer}
LTOF := TOF              {mark new frame}
TOF := TOF + 1
M(TOF) := MRP           {push parent frame pointer}
TOF := TOF + 1
if tag[CC] is last call {return pointer}
then
  M(TOF) := null ptr
else
  M(TOF) := ptr to next call
TOF := TOF + 1
if NC is null ptr
then
  TOF := TOF + 2         {not a backtrack frame}
else
  TOF := TOF + 2         {this is a backtrack frame}
  M(TOF) := NC
  TOF := TOF + 1
  M(TOF) := MRB
  TOF := TOF + 1
  MRB := LTOF
TOF := TOF + 1 + (VC * 3) {allocate three spaces
                           for each variable}
go to call selection
```

### Backtracking

```
if MRB is nil
then
  output (No)           {done}
  quit
else
  NC := M(LTOF + 3)     {restore pointers}
  CC := M(LTOF + 4)
  MRP := M(LTOF + 1)
  TOF := MRB
  LTOF := M(TOF)
  If TOT > M(TOF + 6)   {discard all frames above
                           and including MRB}
  then                  {if bindings have been
                           made unbind them}
    unbind
  MRB := M(TOF + 5)
go to procedure selection
```

### Unbinding

```
while trail pointer < TOT
do
  M (M (TOT)).tag := unbound tag
  TOT := TOT - 1
return
```

### 5.3 Database Form

The set of rules (procedures) and facts (assertions) that may be useful in solving a user's queries are stored in the computer's database in an efficient, codified form. Conceptually, the form can be represented as a set of linked lists. Each entry in a list has three cells. The first cell contains a pointer to the spelling of name of a set of rules or facts, located in an atom heap. The second cell points to the first rule or fact in the linked list of rules and facts that have that name. The third cell is a collision pointer. It points to the next list whose name hashed to the same value in the hash table.

Figure 5.3 shows a simple, recursive Prolog program consisting of one fact and one query. This program defines what John likes as anything that likes wine. The program also includes the fact that Mary likes wine. The query asks, "What does John like."

---

```
likes (john, X) :- likes (X, wine).  
likes (mary, wine).  
:- likes (john, Y).
```

Figure 5.3 An example Prolog program.

---

Figure 5.4 shows the structure of the database. The procedure name, likes, would be accessed through a hash table. The hashed value of likes would point to a linked



list of the names that hash to that value. The entry in the list for likes points to the first occurrence of likes in the facsimile area. The rules and facts in the facsimile area are linked by pointers to the following rule or fact with the same name. Figure 5.5 shows the codified facsimile of the example program. The index numbers that appear along side the entries in figure 5.5 are used in the discussion of the execution of this program in Chapter six.

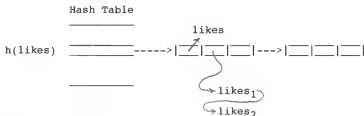


Figure 5.4 Database structure

---

Figure 5.6 shows a more comprehensive example program. This program describes the notion of sisterhood (in at least a limited sense). The program contains a rule which states that one person is the sister of another if the first person is female and the two people have the same mother. The example program contains a second rule stating that two people have the same mother if the first person's mother is the same as the second person's mother. The program then contains a set of facts about females and mothers. The codified facsimile of this example is shown in figure 5.7.

---

Index	Identifier	Tag	Value
1	Next candidate	pointer	index 8
2	Variable Count	procedure	1
3	Atom	atom	----->john
4	Variable	unbound	0
5	Call	last call	h(likes)
6	Variable	unbound	0
7	Atom	atom	----->wine
	End	null pointer	/
		.	
8	Next candidate	null pointer	/
9	Variable Count	assertion	0
10	Atom	atom	----->mary
11	Atom	atom	----->wine
	End	null pointer	/
		.	
12	Goal	null pointer	/
13	Variable Count	procedure	1
14	Call	last call	h(likes)
15	Atom	atom	----->john
16	Variable	unbound	_1

Figure 5.5 Codified facsimile of an example program.

---

The execution of this program will be detailed in Chapter six showing the state of the data structures kept by the interpreter and how the tags of Chapter four facilitate the process.

---

```

sister-of (X,Y) :- female (X), same-mother (X,Y).
same-mother (X,Y) :- mother (X,Z), mother (Y,Z).
female (sue).
female (diane).
mother (diane,sue).
mother (dan,sue).
mother (david,sue).
:- sister-of (diane,A).

```

Figure 5.6 Sisterhood example.

---

Index	Identifier	Tag	Value
1	Next candidate	null pointer	/
2	Variable Count	procedure	2
3	Variable	unbound	-0
4	Variable	unbound	-1
5	Call	procedure	$\bar{h}$ (female)
6	Variable	unbound	0
7	Call	last call	$\bar{h}$ (same-mother)
8	Variable	unbound	-0
9	Variable	unbound	-1
	End	null pointer	7
		.	
		.	
10	Next candidate	pointer	index 13
11	Variable Count	assertion	0
12	Atom	atom	----->sue
	End	null pointer	/
		.	
		.	
13	Next candidate	null pointer	/
14	Variable Count	assertion	0
15	Atom	atom	----->diane
	End	null pointer	/
		.	
		.	

Index	Identifier	Tag	Value
16	Next candidate	null pointer	/
17	Variable Count	procedure	2
18	Variable	unbound	-2
19	Variable	unbound	-3
20	Call	procedure	h(mother)
21	Variable	unbound	-2
22	Variable	unbound	-4
23	Call	last call	h(mother)
24	Variable	unbound	-3
25	Variable	unbound	-4
	End	null pointer	7
	.	.	.
26	Next candidate	pointer	index 30
27	Variable Count	assertion	0
28	Atom	atom	----->diane
29	Atom	atom	----->sue
	End	null pointer	/
	.	.	.
30	Next candidate	pointer	index 34
31	Variable Count	assertion	0
32	Atom	atom	----->dan
33	Atom	atom	----->sue
	End	null pointer	/
	.	.	.
34	Next candidate	pointer	index 38
35	Variable Count	assertion	0
36	Atom	atom	----->david
37	Atom	atom	----->sue
	End	null pointer	/
	.	.	.
38	Goal	null pointer	/
39	Variable Count	procedure	1
40	Call	last call	h(sister-of)
41	Atom	atom	----->diane
42	Variable	unbound	-5

Figure 5.7 Codified facsimile of the sisterhood example program.

## 6.0 Demonstration of Control Flow

Previous chapters of this report have outlined the nature of Prolog programs, the form of Prolog databases and data structures, and proposed a set of tags to implement an interpreter efficiently. This chapter details the execution of two example Prolog programs. The examples were listed and discussed in section 5.3, immediately preceding. The examples show what happens to all of the elements that have been introduced, namely the frame and trail stacks and the global registers, as the interpreter executes the programs. The examples also show how the tags facilitate the execution. The control strategy is detailed in the algorithms presented in Section 5.2.

### 6.1 Likes: A Simple Example

The example program of figure 5.3 is a very short, recursive program whose query asks about what John likes. The rule in the database states that John likes anything that likes wine. The fact in the database states that Mary likes wine. The codified facsimile of the database is shown in figure 5.5. While this is a short example, it

demonstrates all of the important features of the control program, namely call and procedure selection, variable binding and unbinding, backtracking, the function of the frame and trail stacks and how Prolog handles recursion. Snapshots of the frame stack and global registers are included at important points in the execution.

The execution begins with the initialization routine. The initialization routine sets the current procedure pointer to the codified facsimile of the main goal, index 12 in figure 5.5. Then the current procedure pointer is incremented to find the variable count for the procedure. A frame is created for the goal procedure with a null parent pointer, and null previous frame pointer. The variables of the call are pushed on the stack and two extra spaces are allocated for the environment and value pointers for each variable. The state of the frame stack after initialization is shown in figure 6.1. The variable is shown with its tag and value fields separated by a vertical bar. The value pointer is the next entry above the variable, and the environment pointer is the next entry. Throughout this example, the frame stack will grow upward.

The state of the global registers is shown in figure 6.2 below. The frame and trail stack pointers are obvious and will be omitted for brevity.

---

Frame 1 for	Null ptr.		/
	Null ptr.		/
:- likes(john, Y)	UNBOUND		Y
	Return pointer = null		
	parent pointer = null		
	previous frame = null		

---

Frame Stack

Figure 6.1 Frame stack after initialization.

---

Current Call	nil
Current Procedure	main goal, index 13
Next Candidate	null
Most Recent Parent	null
Most Recent Backtrack	null

Figure 6.2 Global registers after initialization.

---

When the initialization is finished, control passes to the call selection routine. Call selection checks the tag of the current procedure, index 13, and finds that it is a procedure. Then the call is found at index 14 and current call points there. Because the current procedure was a call rather than an assertion, the current frame is a parent frame and most recent parent is set to the current frame. The current call, using the hash table, points to the linked list of procedures and assertions that start with like. The next candidate is pointed to by the procedure pointer of the linked list. The next candidate pointer takes the value of

this procedure pointer. Then the next candidate pointer is pointing to index 1. The control then jumps to the procedure selection routine.

Procedure selection begins by saving the top of trail and current call pointers in the second frame of the stack. This is done because an unsuccessful unification attempt will change those pointers and they will need to be restored. Procedure selection then uses the next candidate pointer to get a current procedure. Upon entry to the procedure selection routine, current procedure still points to index 13. Procedure selection changes current procedure to the location one beyond the place next candidate is pointing. Current procedure then points to index 2, the variable count of the procedure

```
likes (john,X) :- likes (X,wine).
```

Next candidate is advanced to the next link in the likes list, namely to index 8.

The next step is the unification procedure which attempts to match the parameters of the current call,

```
likes (john,Y)
```

to the parameters of the current procedure

```
likes (john,X)
```

Because they do match, the interpreter goes to the frame creation routine to build a frame for the procedure that just unified with the current call.



At this point in the execution, the next candidate register is not null showing that there is at least one untried candidate for unification with the current call, so the frame that is created in the frame creation routine is a backtrack frame. The state of the frame stack after the frame is created is shown in figure 6.5. Note that the variable in the procedure became bound to the variable in the call during unification. Thus, the value pointer of the variable in frame two points to the variable in frame 1. This binding is recorded by a pointer to the variable, **X**, in frame 2 on top of the trail stack. The previous frame pointer and parent pointer are shown simply as frame 1. This represents the top of frame 1. The trail was empty when unification began, so the trail pointer points to the bottom of the trail, shown as 0. The trail is shown in figure 6.4, and the global registers are shown in figure 6.3.

---

Current Call	index 14
Current Procedure	index 2
Next Candidate	index 8
Most Recent Parent	null
Most Recent Backtrack	frame 2

Figure 6.3 Global registers after creation of second frame.

---

---

1 pointer to X in frame 2

Trail

Figure 6.4 Trail stack after creation of second frame.

---

---

	Null ptr.	/	
	pointer	----->	Y in frame 1
	BOUND-VAR	X	
Frame 2 for	trail pointer	= 0	
	previous backtrack	= null	
likes (john,X)	previous return	= 14	
	next candidate	= 8	
	return pointer	= null	
:- likes (X,wine)	parent frame	= frame 1	
	previous frame	= frame 1	
	Null ptr.	/	
	Null ptr.	/	
Frame 1 for	UNBOUND	Y	
	Return pointer	= null	
:- likes(john, Y)	parent frame	= null	
	previous frame	= null	

---

Frame Stack

Figure 6.5 Frame stack after creation of second frame.

---

Once the frame has been created, control returns back to the call selection routine. This time the current procedure pointer is pointing to index 5. The call that is selected is the first, and only, call in the procedure. This makes the current call

likes (X, wine).

The next candidate is set to the procedure pointed to by the

procedure pointer of the call, index 1. This is an example of a recursive call. Then control passes to procedure selection.

Procedure selection begins as it did before. The current procedure pointer gets set to index 3. The next candidate pointer gets set to index 8. When the unification procedure is called, the current call is

likes (X,wine)

and the current procedure is

likes (john, X).

Unification would bind X to john and X to wine. Because, a variable cannot be bound to two different atoms, the unification fails and a new current procedure must be found. The state of the global registers at this point is shown in figure 6.6, below.

---

Current Call	index 5
Current Procedure	index 1
Next Candidate	index 8
Most Recent Parent	frame 2
Most Recent Backtrack	frame 2

Figure 6.6 Global registers at unification.

---

Procedure selection assigns the index 1 past the next candidate to the current procedure, thus the current procedure gets index 9. Now the current procedure is the last in the list and so the next candidate is null. When unification is called the current call is still

likes (X,wine),

and the current procedure is

likes (mary, wine).

Unification binds **X** to **mary** and notices that the two atoms match. Thus the current procedure unifies with the current call. However, unlike the last time unification succeeded, the current procedure is an assertion. Thus the variable, **Y**, is bound to the atom, **mary**, and the control passes to the frame creation step. This state of the frame stack is shown in figure 6.8. The state of the trail is shown in figure 6.7.

---

2	pointer to Y in frame 1
1	pointer to X in frame 2

---

Trail

Figure 6.7 Trail stack after creation of third frame.

---

After creating a frame the interpreter returns to call selection to see if there are any more calls to be resolved. Call selection determines that the current procedure is an assertion and begins looking for unanswered calls. The return pointers of frames three and two are null pointers, so the interpreter understands that it has arrived at a solution. At this point the interpreter will output the solution and go to the backtrack routine and look for other solutions.

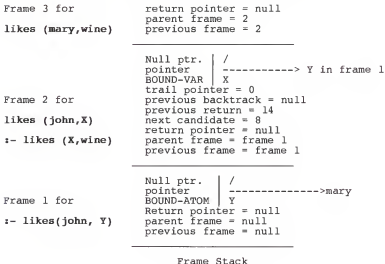


Figure 6.8 Frame stack after creation of third frame.

---

Backtracking restores the pointers that were saved in the backtrack frame, frame two, unbinds any bindings that were made since the creation of frame two, namely the binding of the variable, Y to the atom, mary, and discards the third and second frames. After that, the current call is once again

likes (X,wine).

However, the next candidate pointer is null. At this point control jumps to the procedure selection routine. The

interpreter enters the procedure selection routine with a null next candidate pointer. This invokes backtracking again. What this means is that the interpreter has found all of the solutions to the call,

**likes (X, wine)**

and must look for other procedures to match the previous call,

**likes (john, Y).**

Backtracking resets the current call to the aforementioned previous call, and the next candidate to index 8. The binding of X to Y is removed and the second frame is discarded. At this point the frame stack looks exactly as it did in figure 6.1, just after creation of the first frame.

Procedure selection selects the assertion,

**likes(mary, wine)**

as the current procedure. Of course, the current procedure fails to unify with the current call. This requires further backtracking. The interpreter looks for more unanswered calls and finds none. So, it outputs the answer "No" meaning "no more solutions." There was only one solution to the query, and the interpreter found it.

## 6.2 Sister-of: A More Comprehensive Example

The example program of figure 5.6 is a fairly involved program whose query asks, "Whose sister is Diane?" The

rules in the database state that if Diane is a female, then she is the sister of anyone who has the same mother as her. The codified facsimile of the database is shown in figure 5.7. The execution of this program demonstrates call and procedure selection, variable binding and unbinding, record keeping on the frame and trail stacks, and backtracking both on success and on failure. Snapshots of the frame stack, trail stack and global registers at important points in the execution are included to illustrate the record keeping.

The execution begins, as in the previous example, with initialization. The initialization routine sets the current procedure pointer to the codified facsimile of the main goal, index 38 in figure 5.7. The current procedure is incremented to get the variable count for the current procedure, and a frame is created for the main goal. As always, the main goal has a null parent pointer and null previous frame pointer. The one variable is pushed on the stack and space is allocated for its value and environment pointers. The information is shown in figure 6.9 below. The format is the same as in the previous example with the stack growing up.

The state of the global registers is shown in figure 6.10. The frame and trail stack pointers are obvious and will be omitted for brevity.

---

Frame 1 for

**:- sister-of(diane,A)**

---

Null ptr.		/
Null ptr.		/
UNBOUND		A
Return pointer	=	null
parent frame	=	null
previous frame	=	null

---

Frame Stack

Figure 6.9 Frame stack after initialization.

---

---

Current Call	nil
Current Procedure	main goal
Next Candidate	null
Most Recent Parent	null
Most Recent Backtrack	null

Figure 6.10 Global registers after initialization.

---

After initialization, the interpreter performs call selection. Call selection checks the tag of the current procedure (index 39) and finds that it is a procedure. Because the current procedure was a call rather than an assertion, the current frame is a parent frame and the most recent parent register is set to the current frame. The current call points to the linked list of procedures and assertions that start with

**sister-of.**

The next candidate is pointed to by the procedure pointer of the linked list. The next candidate pointer takes the value



of this procedure pointer. The next candidate pointer is pointing to index 1. The control then proceeds to the procedure selection routine.

Procedure selection begins by saving the top of trail and current call pointer in the second frame of the stack as in the previous example. Current procedure then takes the value of the next candidate, index 1, and the next candidate points to the next member of the linked list of candidates. In this case, there are no more candidates beginning with **sister-of**, so the next candidate is the null pointer. Note that this is done simply by assigning the value field of the previous next candidate to the new next candidate.

Procedure selection calls the unification procedure to see if the current procedure matches the current call and to bind variables if necessary. The unification procedure, then, tries to unify the current procedure,

**sister-of (X,Y),**

to the current call,

**sister-of (diane, A).**

Unification succeeds, and **X** gets bound to **diane** and **Y** gets bound to **A**. Successful unification is analogous to a maze solver choosing to turn at an intersection. There is no guarantee that that turn will lead to the solution, so the maze solver must keep track of the state of his search before making the turn in case it becomes necessary to return to that point and start over. The interpreter saves

the state of the search in the frame stack by creating a new frame.

The frame creation routine saves a pointer to the previous frame, a pointer to the parent frame, and the return pointer in all cases. If there were untried candidates (i.e. the next candidate pointer was not null) then the frame is a backtrack frame and more information is stored. The next candidate pointer is null at this point, so just the three pointers are saved. Above the pointers in the frame, the variable information is stored. The previous frame pointer points to the top of frame 1. The parent of the current procedure is the main goal, so the parent frame pointer also points to frame 1. The return pointer points to the next unresolved call, the call following the current call. In this case the return pointer is null. In the variable space, the X and Y are stored with the information that X is bound to the atom diane and Y is bound to the variable A in the first frame. The state of the frame stack is shown in figure 6.12 below. The state of the global registers is shown in figure 6.11, and the trail stack is shown in figure 6.13.

---

Current Call	sister-of (diane,A)
Current Procedure	sister-of (X,Y)
Next Candidate	null
Most Recent Parent	null
Most Recent Backtrack	null

Figure 6.11 Global registers after frame 2.

---

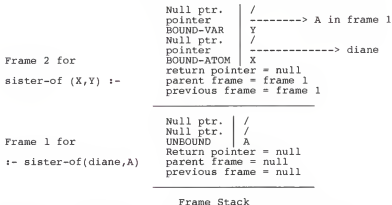


Figure 6.12 Frame stack after creation of second frame.

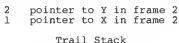


Figure 6.13 Trail stack after frame 2.

---

Once the record keeping has been done, the maze solver can continue down the path he has chosen. Likewise, the interpreter proceeds from frame creation to the search for a new call. The new call is the first call of the current procedure,

**female (X),**

which is located at index 5 in the codified facsimile. The

next candidate is set based on the pointer from the current call. In this case, the next candidate,

female (sue),

is located at index 10.

The procedure selection routine sets the current procedure to the value of the next candidate,

female (sue).

The next candidate is set to the next member of the candidate list,

female (diane),

at index 13. Procedure selection then calls unification to unify the current procedure,

female (sue),

with the current call,

female (X).

Because X is bound to diane and a variable cannot be bound to two different atoms, the unification fails. The state of the registers at this point is shown in figure 6.14 below.

---

Current Call	female (X) {X is bound to diane}
Current Procedure	female (sue)
Next Candidate	female (diane)
Most Recent Parent	frame 2
Most Recent Backtrack	null

Figure 6.14 Global registers at unification.

---

When the current procedure fails to unify with the current call, the procedure selection routine looks for a

new current procedure. The current procedure gets the value of the next candidate,

female (diane),

at index 13, and the next candidate gets the next member of the list of candidates, the null pointer. Then unification is called again and this time succeeds. However, this unification did not involve binding any new variables.

The frame that is created as a result of this unification is very simple. The next candidate pointer is null, so the frame is not a backtrack frame. The current procedure was an assertion, so there are no variables to store. Therefore, only three pointers will be stored. The previous frame is frame 2, the parent frame is frame 2, and the return pointer is the next call, index 7. The state of the frame stack is shown in figure 6.16 below. The state of the global registers is shown in figure 6.15. The trail stack is unchanged.

---

Current Call	female (X)
Current Procedure	female (diane)
Next Candidate	null
Most Recent Parent	frame 2
Most Recent Backtrack	null

Figure 6.15 Global registers after frame 3.

---

---

Frame 3 for  
female (diane).

---

return pointer = index 7  
parent frame = frame 2  
previous frame = frame 2

---

Frame 2 for  
sister-of (X,Y) :-

Null ptr.		/
pointer		-----> A in frame 1
BOUND-VAR		Y
Null ptr.		/
pointer		-----> diane
BOUND-ATOM		X
return pointer		= null
parent frame		= frame 1
previous frame		= frame 1

---

Frame 1 for  
:- sister-of(diane,A)

Null ptr.		/
Null ptr.		/
UNBOUND		A
Return pointer		= null
parent frame		= null
previous frame		= null

---

Frame Stack

Figure 6.16 Frame stack after creation of the third frame.

---

As before, frame creation is followed by the selection of another call. The new current call is the call following female (X), namely the call at index 7,

same-mother (X,Y).

This current call has the tag last call signifying that if the current call is satisfied, its parent call will be satisfied. The next candidate is set by call selection to index 16,

same-mother (X,Y) :-.

Procedure selection assigns the value of the next candidate to the current procedure. The next candidate, pointed to by index 16, is the null pointer. Unification succeeds in unifying the current procedure and current call, and control passes again to frame creation.

The next candidate pointer is, again, a null pointer, so the frame is not a backtrack frame. The fourth frame is created with the previous frame pointer pointing to frame three, the parent frame is frame two, and the return pointer is null. The state of the frame stack is shown in figure 6.19. The global registers and the trail stack are detailed in figures 6.17 and 6.18 respectively.

---

Current Call	same-mother (X,Y)
Current Procedure	same mother (X,Y)
Next Candidate	null
Most Recent Parent	frame 2
Most Recent Backtrack	null

Figure 6.17 Global registers after frame 4.

---

---

4	pointer to Y in frame 4
3	pointer to X in frame 4
2	pointer to Y in frame 2
1	pointer to X in frame 2

Trail Stack

Figure 6.18 Trail stack after frame 4.

---

---

Frame 4 for same-mother (X,Y)	<pre> null ptr        / pointer         -----&gt; Y in frame 2 BOUND-VAR      Y null ptr        / pointer         -----&gt; X in frame 2 BOUND-VAR      X return pointer = null parent frame = frame 2 previous frame = frame 3 </pre>
Frame 3 for female (diane).	<pre> return pointer = index 7 parent frame = frame 2 previous frame = frame 2 </pre>
Frame 2 for sister-of (X,Y)	<pre> Null ptr.      / pointer        -----&gt; A in frame 1 BOUND-VAR     Y Null ptr.      / pointer        -----&gt; diane BOUND-ATOM    X return pointer = null parent frame = frame 1 previous frame = frame 1 </pre>
Frame 1 for :- sister-of(diane,A)	<pre> Null ptr.      / Null ptr.      / UNBOUND       A Return pointer = null parent frame = null previous frame = null </pre>

---

Frame Stack

Figure 6.19 Frame stack after creation of fourth frame.

---

Once again the call selection routine is executed to find the next call to address. This time the current call is the first call of

same-mother (X,Y),



namely

`mother (X,Z),`

at index 20. The next candidate is index 26,

`mother (diane,sue).`

The procedure selection routine sets the current procedure to index 26, and the next candidate to index 30,

`mother (dan,sue).`

Unification attempts to unify

`mother (X,Z)`

with

`mother (diane,sue),`

while X is bound to the atom, diane, and Z is unbound. This succeeds and binds Z to sue.

The frame creation routine finds the next candidate pointer not null, and the frame that is created is the first example of a backtrack frame. The previous frame is, of course, frame 4, and the parent frame, from the most recent parent register, is also frame 4. The return pointer is the next call,

`mother (Y,Z),`

located at index 23. Four other pointers are also stored in a backtrack frame. The next candidate is

`mother (dan,sue).`

The previous return pointer is simply the current call,

`mother (X,Z).`

The previous backtrack is null, because this is the first

backtrack frame. The trail pointer points to the top of the trail in figure 6.18, index 4. All of this information and the variable bindings is shown in figure 6.20 below. The state of the global registers and trail follows in figures 6.21 and 6.22.

---

<p>Frame 5 for mother (X,Z)</p>	<pre> Null ptr.   / pointer     -----&gt; sue BOUND-ATOM   Z Null ptr.   / pointer     -----&gt; X in frame 4 BOUND-VAR   X trail ptr.   = 4 previous backtrack = null previous return = index 20 next candidate = index 30 return pointer = index 23 parent frame = frame 4 previous frame = frame 4 </pre>
<hr/>	
<p>Frame 4 for same-mother (X,Y)</p>	<pre> null ptr   pointer     -----&gt; Y in frame 2 BOUND-VAR   Y null ptr    / pointer     -----&gt; X in frame 2 BOUND-VAR   X return pointer = null parent frame = frame 2 previous frame = frame 3 </pre>
<hr/>	
<p>Frame 3 for female (diane).</p>	<pre> return pointer = index 7 parent frame = frame 2 previous frame = frame 2 </pre>
<hr/>	
<p>Frame 2 for sister-of (X,Y)</p>	<pre> Null ptr.   / pointer     -----&gt; A in frame 1 BOUND-VAR   Y Null ptr.   / pointer     -----&gt; diane BOUND-ATOM   X return pointer = null parent frame = frame 1 previous frame = frame 1 </pre>

---

```

Frame 1 for
:- sister-of(diane,A)
Null ptr. | /
Null ptr. | /
UNBOUND  | A
Return pointer = null
parent frame = null
previous frame = null

```

---

Frame Stack

Figure 6.20 Frame stack after creation of the fifth frame.

---



---

```

Current Call           mother (X,Z)
Current Procedure      mother (diane, sue)
Next Candidate        mother (dan,sue)
Most Recent Parent    frame 4
Most Recent Backtrack frame 5

```

Figure 6.21 Global registers after frame 5.

---



---

```

6 pointer to Z in frame 5
5 pointer to X in frame 5
4 pointer to Y in frame 4
3 pointer to X in frame 4
2 pointer to Y in frame 2
1 pointer to X in frame 2

```

Trail Stack

Figure 6.22 Trail stack after frame 5.

---

After creation of frame 5, the call selection routine sets the current call to the last call in the same-mother rule, index 23,

mother (Y,Z).

The next candidate is

**mother (diane, sue),**

at index 26. Procedure selection moves the next candidate into the current procedure and sets the next candidate to index 30,

**mother (dan, sue).**

Unification compares

**mother (Y,Z)**

and

**mother (diane,sue).**

The variable, *Y*, is initially bound to the unbound variable, *A*, in frame 1, and the variable, *Z*, is bound to the atom, *sue*, in frame 5. Thus unification succeeds, binding the variable, *A*, to the atom, *diane*. The frame that is created is a backtrack frame and is shown in figure 6.23. The global registers and trail follow in figures 6.24 and 6.25.

Frame 6 for  
mother (Y,Z)

---

Null ptr.	/
pointer	-----> Z in frame 5
BOUND-VAR	Z
Null ptr.	/
pointer	-----> Y in frame 4
BOUND-VAR	Y
trail ptr.	= 6
previous backtrack	= frame 5
previous return	= index 20
next candidate	= index 30
return pointer	= null
parent frame	= frame 4
previous frame	= frame 5

---

Frame 5 for mother (X,Z)	Null ptr.   / pointer   -----> sue BOUND-ATOM   Z Null ptr.   / pointer   -----> X in frame 4 BOUND-VAR   X trail ptr. = 4 previous backtrack = null previous return = index 20 next candidate = index 30 return pointer = index 23 parent frame = frame 4 previous frame = frame 4
<hr/>	
Frame 4 for same-mother (X,Y)	null ptr   / pointer   -----> Y in frame 2 BOUND-VAR   Y null ptr   / pointer   -----> X in frame 2 BOUND-VAR   X return pointer = null parent frame = frame 2 previous frame = frame 3
<hr/>	
Frame 3 for female (diane).	return pointer = index 7 parent frame = frame 2 previous frame = frame 2
<hr/>	
Frame 2 for sister-of (X,Y)	Null ptr.   / pointer   -----> A in frame 1 BOUND-VAR   Y Null ptr.   / pointer   -----> diane BOUND-ATOM   X return pointer = null parent frame = frame 1 previous frame = frame 1
<hr/>	
Frame 1 for :- sister-of(diane,A)	Null ptr.   / pointer   -----> diane BOUND -ATOM   A Return pointer = null parent frame = null previous frame = null

Figure 6.23 Frame stack after creation of the sixth frame.

---

Current Call	mother (Y,Z)
Current Procedure	mother (diane, sue)
Next Candidate	mother (dan,sue)
Most Recent Parent	frame 4
Most Recent Backtrack	frame 6

Figure 6.24 Global registers after frame 6.

---

---

9	pointer to A in frame 1
8	pointer to Z in frame 6
7	pointer to Y in frame 6
6	pointer to Z in frame 5
5	pointer to X in frame 5
4	pointer to Y in frame 4
3	pointer to X in frame 4
2	pointer to Y in frame 2
1	pointer to X in frame 2

Trail Stack

Figure 6.25 Trail stack after frame 6.

---

As always, call selection is entered after frame creation, however this time the current procedure is an assertion, so there are no new calls to pursue, and there are no more unresolved calls to answer. When that happens the interpreter understands that the query has been successfully answered. The solution,

A = diane,

is printed for the user, and backtracking is initiated to find more solutions.

The solution that diane is the sister of diane may come as a surprise to some. We do not normally think of a person

being her own sister. However, the program stated that one person is the sister of another if one is a female and they have the same parents. The program did not state that the people had to be unique. Naturally, a rule could be added to preclude this solution, but it is more informative to consider changing the query to prevent the trivial solution from being chosen. The query could be written using the built in predicate, `/=`, meaning not equal to, as follows:

```
:- sister-of (diane,A), A /= diane.
```

If the query had been written that way, the call selection routine after frame 6 was created would have found one more call to answer:

```
A /= diane.
```

That call would of course fail given that `A` is instantiated to `diane`. This failure would initiate backtracking just as the success of the last call did with the initial query. The only difference is that the result,

```
A = diane,
```

would not be printed.

The backtracking routine behaves the same regardless of whether it was entered after a success, as with the initial query, or on failure, as with the modified query. In either case there is a backtrack frame on the stack signifying that there are untried solution paths. The job of the backtracking routine is to restore the state of the database

as it was when the last backtrack frame was created. This involves restoring the next candidate register from the next candidate entry in the frame, restoring the current call from the previous return entry in the frame, restoring the most recent parent register from the parent frame pointer entry of the frame, and restoring the most recent backtrack register from the previous backtrack entry in the frame. Then the frame is discarded by setting the top of frame pointer to the position recorded in the most recent backtrack register. This action restores all of the global registers, except current procedure, and the frame stack to the state when the backtrack frame was created. (Note that the current procedure register is left unchanged during backtracking and will not be updated until the next call to procedure selection.)

All that is left is unbinding the variables that were bound between the creation of the backtrack frame and the initiation of backtracking. Those variable bindings are recorded on the trail stack between the top of the trail stack, position 9, and the top of the trail stack when the backtrack frame was created, position 6, recorded in the trail pointer entry of the backtrack frame. The states of the frame stack and trail stack are exactly as they were after the creation of the fifth frame (see figures 6.20 and 6.22). The global registers are detailed in figure 6.26 below.



---

Current Call	mother (Y,Z)
Current Procedure	mother (diane, sue)
Next Candidate	mother (dan,sue)
Most Recent Parent	frame 4
Most Recent Backtrack	frame 5

Figure 6.26 Global registers after backtracking.

---

After backtracking, all of the global registers are restored to the state of the search before the unification that lead to backtracking. Control then passes to the procedure selection routine. As always, the procedure that is selected is the next candidate, and the next candidate is chosen from the list of candidate procedures. This makes the current procedure

`mother(dan, sue),`

and the next candidate is

`mother (david, sue).`

Unification is then called to match the current procedure with the current call,

`mother (Y,Z),`

where Y is bound to an unbound variable, A, and Z is bound to the atom, sue.

Unification succeeds, binding A to the atom, dan. There is an untried candidate, so the frame that is created is a backtrack frame. The states of the frame stack, global registers, and trail stack are shown in figures 6.27 through 6.29.

---

Frame 6 for  
mother (Y,Z)

---

```
Null ptr. | /  
pointer   | -----> Z in frame 5  
BOUND-VAR | Z  
Null ptr. | /  
pointer   | -----> Y in frame 5  
BOUND-VAR | Y  
trail ptr. = 6  
previous backtrack = frame 5  
previous return = index 20  
next candidate = index 34  
return pointer = null  
parent frame = frame 4  
previous frame = frame 5
```

---

Frame 5 for  
mother (X,Z)

---

```
Null ptr. | /  
pointer   | -----> sue  
BOUND-ATOM | Z  
Null ptr. | /  
pointer   | -----> X in frame 4  
BOUND-VAR | X  
trail ptr. = 4  
previous backtrack = null  
previous return = index 20  
next candidate = index 30  
return pointer = index 23  
parent frame = frame 4  
previous frame = frame 4
```

---

Frame 4 for  
same-mother (X,Y)

---

```
null ptr | /  
pointer  | -----> Y in frame 2  
BOUND-VAR | Y  
null ptr | /  
pointer  | -----> X in frame 2  
BOUND-VAR | X  
return pointer = null  
parent frame = frame 2  
previous frame = frame 3
```

---

Frame 3 for  
female (diane).

---

```
return pointer = index 7  
parent frame = frame 2  
previous frame = frame 2
```

---

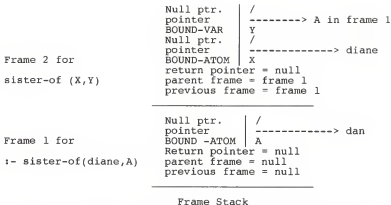


Figure 6.27 Frame stack after creation of the sixth frame.

---

Current Call	mother (Y,Z)
Current Procedure	mother (dan,sue)
Next Candidate	mother (david,sue)
Most Recent Parent	frame 4
Most Recent Backtrack	frame 6

Figure 6.28 Global registers after frame 6.

---

9	pointer to A in frame 1
8	pointer to Z in frame 6
7	pointer to Y in frame 6
6	pointer to Z in frame 5
5	pointer to X in frame 5
4	pointer to Y in frame 4
3	pointer to X in frame 4
2	pointer to Y in frame 2
1	pointer to X in frame 2

Trail Stack

Figure 6.29 Trail stack after frame 6.

---

After frame creation is call selection. Just as before, there are no more calls, so the interpreter outputs the solution,

A = dan.

Control proceeds to backtracking as before and the state of the search is exactly the same as it was after the last backtracking except that the next candidate is now index 34,

mother (david, sue).

Procedure selection assigns that candidate to current procedure, and the next candidate is the null pointer. Unification succeeds and a new frame 6 is built. However, the new frame 6 is not a backtrack frame. This state is recorded in figures 6.30 through 6.32.

---

Current Call	mother (Y,Z)
Current Procedure	mother (david,sue)
Next Candidate	null
Most Recent Parent	frame 4
Most Recent Backtrack	frame 5

Figure 6.30 Global registers after frame 6.

---

---

9	pointer to A in frame 1
8	pointer to Z in frame 6
7	pointer to Y in frame 6
6	pointer to Z in frame 5
5	pointer to X in frame 5
4	pointer to Y in frame 4
3	pointer to X in frame 4
2	pointer to Y in frame 2
1	pointer to X in frame 2

Trail Stack

Figure 6.31 Trail stack after frame 6.

---

---

Frame 6 for mother (Y,Z)	Null ptr.   / pointer   -----> Z in frame 5 BOUND-VAR   Z Null ptr.   / pointer   -----> Y in frame 4 BOUND-VAR   Y return pointer = null parent frame = frame 4 previous frame = frame 5
-----------------------------	---

---

Frame 5 for mother (X,Z)	Null ptr.   / pointer   -----> sue BOUND-ATOM   Z Null ptr.   / pointer   -----> X in frame 4 BOUND-VAR   X trail ptr. = 4 previous backtrack = null previous return = index 20 next candidate = index 30 return pointer = index 23 parent frame = frame 4 previous frame = frame 4
-----------------------------	---

---

Frame 4 for same-mother (X,Y)	null ptr   / pointer   -----> Y in frame 2 BOUND-VAR   Y null ptr   / pointer   -----> X in frame 2 BOUND-VAR   X return pointer = null parent frame = frame 2 previous frame = frame 3
----------------------------------	---

---

Frame 3 for female (diane).	return pointer = index 7 parent frame = frame 2 previous frame = frame 2
--------------------------------	--

---

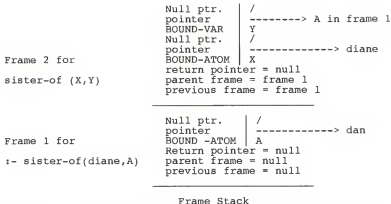


Figure 6.32 Frame stack after creation of the sixth frame.

Call selection will output the solution

**A = david.**

Then control passes to backtracking. The most recent backtrack point is frame 5, so the database is restored to its state before frame 5 was created. Frames 5 and 6 are deleted, the variables pointed to by the trail entries 5 through 9 are changed to unbound, and those trail entries are removed. The frame and trail stacks look like they did after creation of frame 4 (see figures 6.18 and 6.19). The global registers are detailed in figure 6.33.

The current call reverts to

**mother (X,Z)**

where the variable, X, is bound to the atom, diane, and the variable, Z, is bound to the atom, sue. The procedure

---

Current Call	mother (X,Z)
Current Procedure	mother (david,sue)
Next Candidate	mother (dan,sue)
Most Recent Parent	frame 4
Most Recent Backtrack	null

Figure 6.33 Global registers after last solution.

---

selection routine selects

mother (dan,sue),

which fails to unify, and then selects

mother (david,sue),

which also fails to unify. Then the list of candidates is exhausted and control passes to backtracking.

When backtracking is entered with a null most recent backtrack pointer, the interpreter knows that there are no more solutions, and it outputs

No.

At that point the execution of the query

sister-of (diane, A)

is completed. All of the solutions were found and output as follows:

A = diane

A = dan

A = david

No.

## 7.0 Conclusions

The goal of this report was to propose a tagged computer architecture that would lend itself to an efficient implementation of the Prolog computer language. The inference mechanism of a Prolog interpreter was described and a set of tags based on the Prolog data structures and the control algorithm of the interpreter was proposed.

The computer described in this report would be capable of reasonably fast execution. However, for a substantial increase in processing speed, the parallelism of Prolog must be exploited. This is an appropriate area for further research.



### References

- [1] Meyers, Glenford J. Advances in Computer Architecture.  
New York: John Wiley & Sons, 1978
- [2] Feustel, Edward A. "On the Advantages of Tagged  
Architecture." IEEE Transactions on Computers,  
vol. C-22, no. 7, July 1973.
- [3] Clocksin, W. F. and C. S. Mellish. Programming in  
Prolog 2nd ed. Berlin: Springer-Verlag, 1984.
- [4] Rodenbaugh, David J. A Control Strategy for a Prolog  
Interpreter, A Master's Report. Kansas State  
University, Manhattan, Kansas, 1985.
- [5] Rich, Elaine. Artificial Intelligence. New York:  
McGraw-Hill, Inc., 1983.
- [6] Lloyd, J. W. Foundations of Logic Programming.  
Berlin: Springer-Verlag, 1984.
- [7] Hogger, Christopher. Introduction to Logic  
Programming. London: Academic Press, 1984.

INVESTIGATION OF  
A TAGGED COMPUTER ARCHITECTURE  
FOR THE PROLOG LANGUAGE

by

DANIEL R. MEIGS

B.S., Kansas State University, 1985

---

AN ABSTRACT OF A MASTER'S REPORT  
submitted in partial fulfillment of the  
requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY  
Manhattan, Kansas

1988

### Abstract

Artificial Intelligence programming is done primarily in LISP and Prolog. Since 1960 effort has been expended toward making a dedicated LISP machine. However, only in the last decade has research been done on a computer dedicated to Prolog. This report focuses on the advantages of, and implementation of, a tagged architecture tailored to Prolog. While the emphasis in this report is on Prolog, a tagged architecture is a very flexible design approach. The concepts in this report are valid for other applications where a computer should be tailored to the task rather than the task to a computer.