

/ A CONTROL STRATEGY FOR A PROLOG INTERPRETER /

by

DAVID J. RODENBAUGH

B.S., Kansas State University, 1984

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

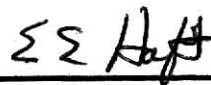
MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:



Major Professor

LD
2668
R4
1985
R62
C.2

A11202 964641

TABLE OF CONTENTS

	Page
Abstract	ii
List of Figures	iv
Acknowledgements	vi
I. Introduction	1
II. Overview of the PROLOG Language	3
III. High Level Algorithm and Major Data Structures	7
IV. Codification	10
A. Data Structures	10
B. Codification Algorithms	20
V. Initialization	32
VI. Call Selection	34
VII. Procedure Selection	38
A. Unification	41
B. Variable Binding	47
C. The Trail	51
VIII. Frame Creation	55
IX. Backtracking	59
X. Summary of Inference Engine	61
XI. Conclusions	68
References	69

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

LIST OF FIGURES

Figure	Title	Page
1.	Tree Representation of a Literal	4
2.	High Level Flow of Interpreter Control	9
3.	Example of Tree Structure of a Procedure	11
4.	Example of Final Codified Form of a Procedure	13
5.	Atom Hash Table	14
6.	Example of Variable List Used During Codification	15
7.	Procedure Hash Table and Node Organization	17
8.	Use of Codification Stack	19
9.	Algorithm for Building Binary Tree for a Procedure	21
10.	Linking a Procedure to the List of Procedures with Identical Names	23
11.	Codified Procedure Format	24
12.	Obtaining Top Level Parameters of the Procedure Head	26
13.	Stacking of Structure Information During Linearization of Binary Tree	27
14.	Use of Call Identifier	28
15.	Linearize Tree Flowchart	31
16.	Get First Level Parameters Flowchart	31
17.	Initialization Flowchart	33
18.	Call Selection Flowchart	36
19.	Accessing Procedures Matching Call Name	37
20.	Procedure Selection Flowchart	39

Figure	Title	Page
21.	Recording Pointers in New Frame During Unification . . .	40
22.	Binary Tree Traversal Routine	41
23.	Order of Tree Traversal	42
24.	Matching Variables with Structures	43
25.	Parameter Matching Rules	44
26.	Initializing Pointers for Unification Process	45
27.	Skeleton Pointer Indirection	46
28.	Database for Unification Procedure	48
29.	Frame for Main Goal	49
30.	Frame Stack After One Unification	49
31.	Variables Bound to Structures	50
32.	Frame Stack after all Variables are Instantiated	52
33.	Use of Trail Stack	53
34.	State of Frame Stack Prior to Backtracking	53
35.	Backtrack Frame Format	57
36.	Frame Creation Flowchart	58
37.	Backtracking Flowchart	60
38.	Database and Query for Interpreter Operation	63
39.	Data Structures after Main Goal Frame is Built	64
40.	Frame Stack after Three Frames Built	65
41.	Frame Stack after Backtracking	66
42.	Frame Stack after Goal is Solved	67

ACKNOWLEDGEMENTS

I wish to express to Dr. E. Haft my appreciation for his time, suggestions, and the flexibility he allowed me in selecting this non-traditional topic.

I. INTRODUCTION

Since knowledge representation is a major part of artificial intelligence, an effort started in the 1970's to find ways to represent knowledge in a form compatible with a computer. Conceptual graphs, a form of logic notation, led to the use of logic as a means of representing knowledge in the form of a programming language (1:137). PROLOG is the most popular logic programming language and is based on predicate calculus (first-order logic).

PROLOG gained worldwide attention when the Institute for New Generation Computing Technology (ICOT) of Japan selected PROLOG as the kernel language for their fifth generation computer system (2:160). Although LISP has dominated the attention of research in this country as an artificial intelligence language (3:395), there has been increasing interest in PROLOG since ICOT's announcement.

Interest in PROLOG leads to a search for a machine architecture on which the language will run efficiently. In order to arrive at such an architecture, it is certainly necessary to understand the inference mechanism on which PROLOG is based. This report outlines an algorithm and data structures for a PROLOG interpreter. The algorithm and data structures herein are based upon a high level algorithm by Hogger (4:190). Although a full implementation was not done, approximately 1500 lines of C

code were written to test many of the concepts.

The organization of this report is as follows. Chapter II gives an overview of the PROLOG language, including syntax and semantics. Chapter III introduces the high level algorithm and major data structures the interpreter uses. Chapter IV presents a scheme for codifying the PROLOG database in a form that is compact and conducive to quick unification (pattern matching). It should be mentioned that the literature is sparse for this portion of the interpretive process. Chapters V through IX detail the important steps of the interpreter, with a summary of the process given in Chapter X. Suggestions that merit more research for computer architectures on which to run PROLOG more efficiently are given in Chapter XI.

II. OVERVIEW OF THE PROLOG LANGUAGE

A PROLOG program is a collection of special first order logic clauses called Horn clauses (5:222). Also known as procedures, these clauses are analogous to natural language sentences, a feature that makes PROLOG relatively easy to use. These PROLOG procedures can be used to represent either assertions or rules. The interpreter uses the database of rules and facts to determine if a query made by the user to the database can be deduced using a technique called resolution (6:80). A description of the syntax and semantics of PROLOG, along with an overview of resolution follow in this chapter.

The literal is the fundamental unit of a logic program. A literal has a predicate part and a sequence of zero or more arguments. The number of arguments is referred to as its arity. For example, "a(b,c,d)" is a literal with predicate "a" and arguments "b", "c", and "d" for an arity of three.

The arguments of a literal are also called terms. Terms can be constants, variables, or a compound term. This implies a recursive definition of the language. This allows structured arguments and the possibility to represent the literals as tree structures. An example tree representation for the literal "a(b,c(d,e),f)" is shown in Figure 1.

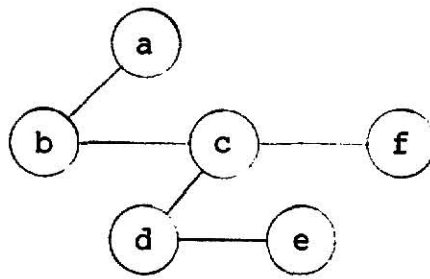


FIGURE 1

Tree Representation of a Literal

The constants of PROLOG are either numbers or atoms, the latter being a sequence of characters starting with a lower case letter. Variables, which must be capitalized in PROLOG, stand for an undetermined object, and are local to a particular procedure.

A procedure consists of a head and body, separated by ":-", the implication symbol. The head is made of zero or one literal. A procedure with no head is called the goal statement or query. There is only one headless procedure in a PROLOG program. The body is comprised of zero or more literals, known as calls. Examples of a headless and a headed procedure are shown below.

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

The semantics of a PROLOG rule is interpreted such that the procedure head is true if all the calls of the body are true. For example, the rule, "today is Monday if yesterday was Sunday", can be represented by:

```
today(monday) :- yesterday (sunday).
```

The assertions of the database are those procedures without calls. A possible assertion to represent is "Yesterday was Sunday".

```
yesterday (sunday).
```

A possible goal statement is the query corresponding to "What is today?"

```
:-today(X).
```

How a query is solved will now be discussed. Resolution, discovered by J. Alan Robinson, is the principle on which the PROLOG interpreter is based. Resolution is an inference rule in which attempts to show that a query is a logical consequence of the database by proving that the negation of that query is inconsistent.

What this amounts to is the following scenario. A match is attempted between the goal and the head of one of the clauses. In order for a successful match to be made, the predicate and arguments must match. A variable will match with anything if it has no value bound to it or has an identical value as the argument in question. Each call of the procedure's body is matched to heads of others procedures in the database. If any of these procedures have calls, all of these calls must also be matched to procedures. This is continued until all literals have been matched, or until no solution can be found. To illustrate, the previous example is used.

```
today (monday) :- yesterday (sunday).  
yesterday (sunday).  
:- today (X).
```

The goal, ":- today (X)", is matched with the head of the first rule, binding the variable "X" to the atom "monday". This leaves one unmatched literal, "yesterday (sunday)". It is matched with the second clause, and the query is solved.

A high level algorithm and associated data structures used by the interpreter will now be described.

III. HIGH LEVEL ALGORITHM AND MAJOR DATA STRUCTURES

This chapter serves to outline the major steps of the algorithm and to introduce the data structures (4:182). Detailed descriptions of each step in the algorithm will be covered in the following chapters, along with a more detailed description of the data structures associated with each step. The data structures that will be introduced are:

- a. codified facsimile area
- b. frame stack
- c. trail stack
- d. global registers
- e. atom heap

The codified facsimile area is an area of memory where the database procedures are stored in an efficient, coded form. This allows for a quicker unification than if the database was stored in its original, textual format.

The frame stack is an area of memory for storing data assignments to variables, and for storing information the interpreter needs to determine the next call to solve. A frame is created for each successful unification (match) of a call with a procedure head. Frames are deleted upon backtracking, a process of undoing the effects of searching a fruitless path.

The trail stack is necessary because of backtracking. When backtracking occurs, all recently created frames up to and inclu-

ding the previous backtrack point are deleted. Any data assignments that were made to variables of earlier frames due to the creation of the more recent frames must be undone. The trail stack keeps track of these variables.

The global pointer registers used include MOST RECENT BACKTRACK, CURRENT PROCEDURE, CURRENT CALL, NEXT CANDIDATE, MOST RECENT PARENT, and TOP OF TRAIL. MOST RECENT BACKTRACK indicates the frame for which untried candidate procedures remain for a call. CURRENT PROCEDURE is a pointer to the location in the facsimile where the procedure being matched to the call to be solved is encoded. CURRENT CALL points to the facsimile of that call. NEXT CANDIDATE points to the next untried procedure in the database which has not been tried in a unification attempt with the current call to be solved. MOST RECENT PARENT indicates which frame the CURRENT CALL resulted from. Finally, TOP OF TRAIL is the trail stack pointer.

The high level flow of the interpreter is shown in Figure 2. The organization and purpose of the data structures just introduced will become more apparent as each step of the interpreter algorithm is described in the following chapters.

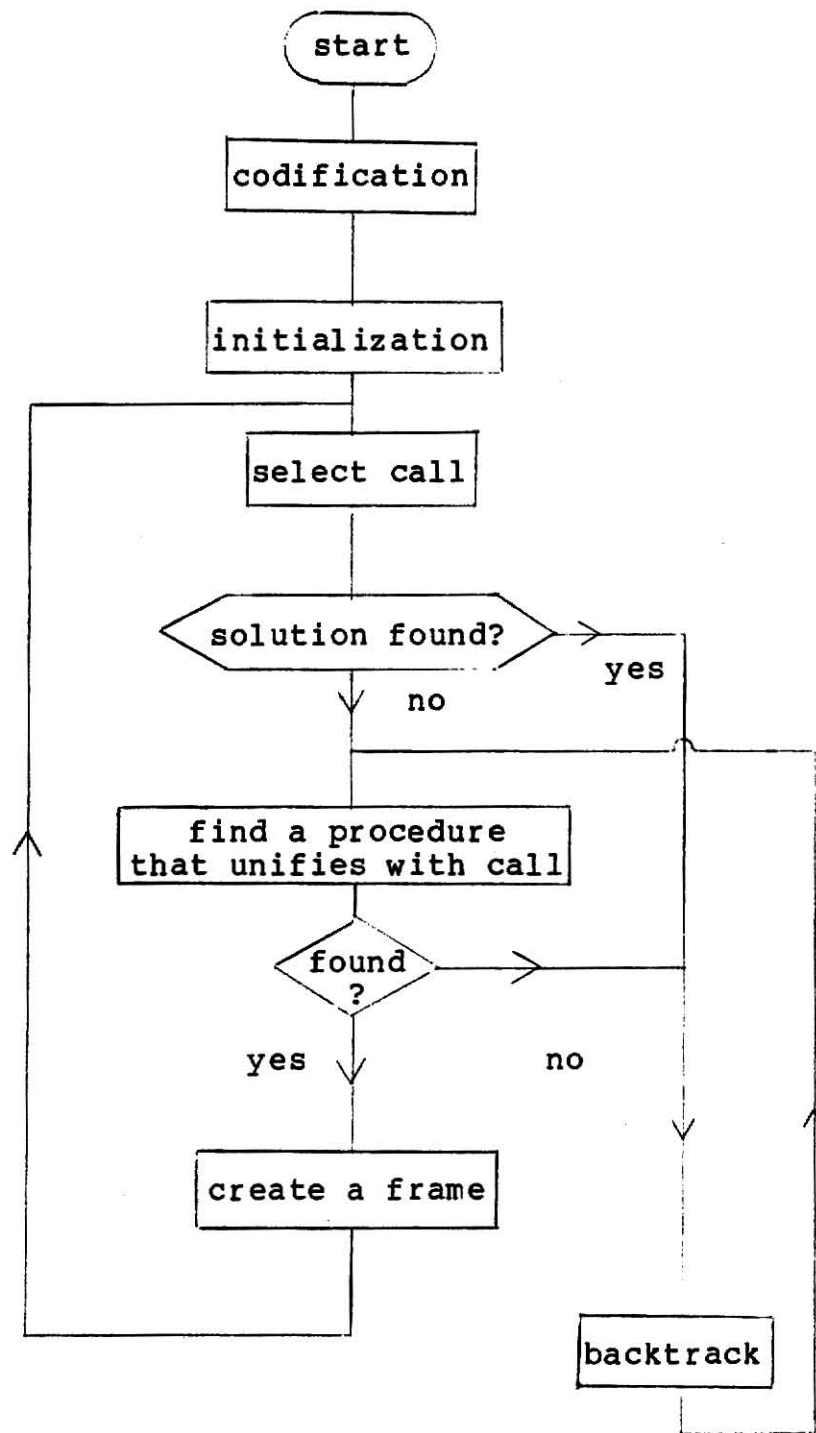


FIGURE 2

High Level Flow of Interpreter Control

IV. CODIFICATION

The database as the user defines it is not in a form conducive to efficient unification. Some preprocessing is done to break the database into tokens, attach a tag to the tokens that describes what they are, and then organize the tagged data in a uniform manner so that the interpreter can easily find the needed information. Because the database statements can be quite complicated as a result of their being comprised of recursive structures, an intermediate form of the statements is produced. Final codification is then done on the intermediate form.

The data structures used by the codification step will be described, followed by the algorithms that operate on them to perform the codification.

Data Structures

The major steps of the codification process deal with several important data structures that will be described. These structures include the following:

- a. binary tree
- b. facsimile area
- c. atom hash table
- d. variable heap
- e. procedure hash table and nodes
- f. codification stack

Binary Tree. Since structures can be represented as a binary tree, this was chosen as the intermediate form to put the statements in. Figure 3 shows an example of a procedure represented in binary tree form. Note the left branch corresponds to going down a level within a structure to obtain the parameters. Likewise, traversing the right branch corresponds to obtaining the next argument of a given level of the structure.

`a(B,c(d,E)) :- g(h).`

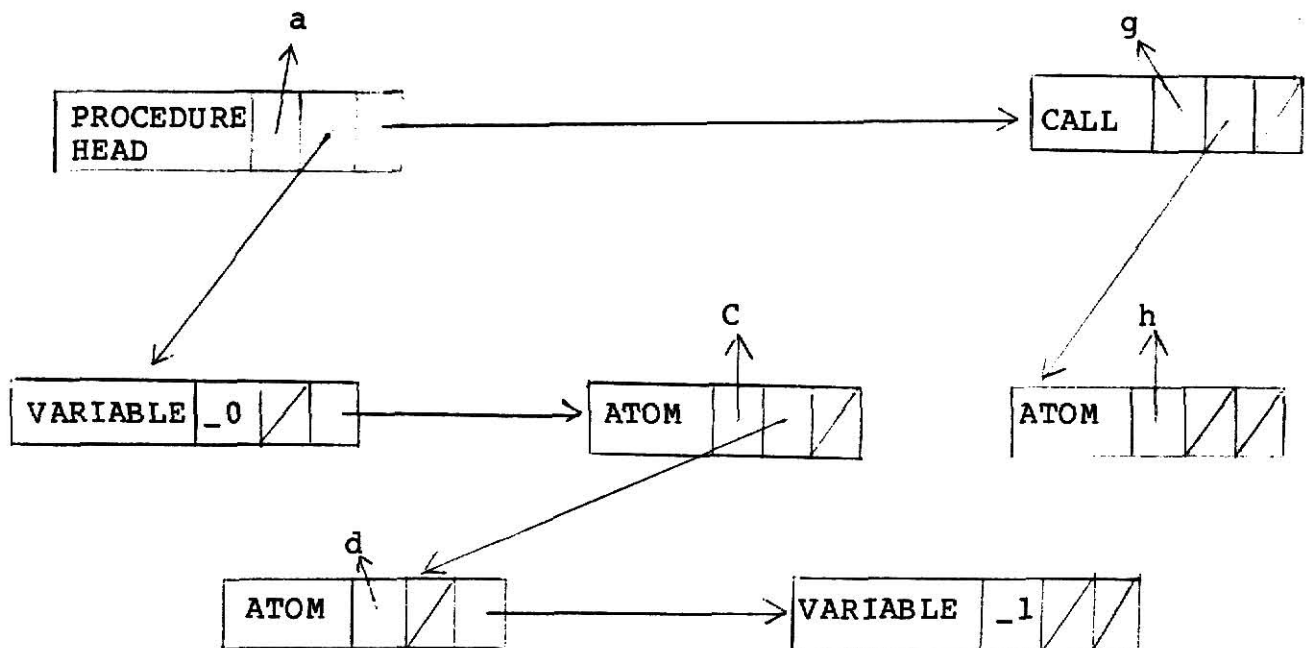


FIGURE 3

Example of Tree Structure of a Procedure

Facsimile Area. The final codification amounts to putting the binary tree in a linear form so that less indirection is used

during unification. Parameters at the same level in the binary tree, connected by the right branch pointer, are in sequential order in the data base. If any of the parameters are themselves structures, then the skeleton of the structure is accessed through the left pointer of the binary tree node. Because successive locations are already used to remove the need for the right pointer in the facsimile, the need for the left pointer remains for the codified facsimile in order to retain the structure information. This pointer is called the skeleton pointer.

Some additional bookkeeping information is also put in the facsimile entry for a procedure. A pointer to the next procedure with the same name is put in the facsimile so that candidate procedures for a call to be solved are easily found during unification. Also a count of the number of variables in the procedure is saved so that the amount of memory necessary for recording their values during unification is easily calculated. The same example above in final form is shown in Figure 4.

The advantages of linearizing the binary tree are a savings of time and memory. The binary tree form requires a left and right pointer for each parameter for the child and next sibling pointers, respectively. With the linear form, no sibling pointer is used. It is assumed that the next sibling is in the next memory location after the current parameter, thus less indirection is necessary. The pointers that do remain are the first child (left subtree) pointers which are called structure skeleton pointers in the literature.

a(B,c(d,E)) :- g(h).

Facsimile Area

NEXT CANDIDATE		
VARIABLE COUNT	2	
VARIABLE	_0	
STRUCTURE		
CALL		
ATOM		→ h
STRUCTURE DEFINITION		←
ATOM		→ c
ATOM		→ d
VARIABLE	_1	

Figure 4

Example of Final Codified Form of a Procedure

Atom Hash Table. All atoms in the data base are stored in an area referred to as the heap when put in codified form (7:135). Each atom is separated by a null character. All entries in the atom heap are unique, regardless of the number of times the atoms appear in the source code. This is done for two reasons. First, there is a savings in memory if one copy of an atom appears in the heap when it may show up numerous times in the database source code. Secondly, during unification it is necessary to determine if two parameters are equal. If distinct copies of the atoms were kept, a character by character

comparison would have to be done. With the unique atom scheme, only one comparison is necessary--that of the pointers to the atoms.

One problem during codification is that of determining if an atom has been used before. If all atoms were in a sorted list, at best it would take approximately $n \log n$ operations, where n is the number of atoms. An alternate method, and that used in this implementation, is the use of a hash table. A hash function is applied to the atom which results in a number used to index the hash table. The entry in the hash table consists of a pointer to a linked list of atom nodes of information. An atom node contains two pieces of information. The first field is a pointer to the atom in the heap area. The second is a link to another atom node in case an atom collided (has the same hash value) with a previous atom. If no collision occurs, the field is nil. An example of a hash table and atom nodes is shown in Figure 5. Here atoms "apple" and "mary" hash to the same value in the hash table. Thus the "mary" atom node is accessed through the collision pointer of the "apple" atom node.

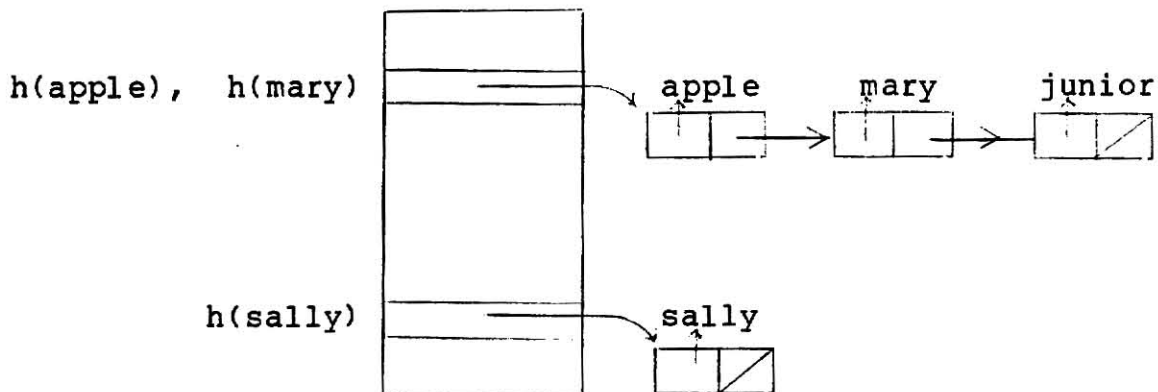


FIGURE 5

Atom Hash Table

Variable List. The variables are not kept in a hash table. This is because variables have a scope of reference only within a procedure. Thus "X" in one procedure is not the same "X" as in another. Instead, variables within a procedure are given an index number starting with zero. Each distinct variable is given a distinct index number. The index number turns out to be much more convenient than the actual variable name or a pointer because any value bound to a variable can easily be found. They are kept in an order determined by their index value in a frame built for a particular call-procedure head unification.

For codification purposes, an array is build with the contents of a particular element location being the actual string representation of the variable. When determining if a variable has been used in the procedure previously, a scan is made of the array. If it is not found, it is put at the end of the list. The entry after the end of the list is a null character. In the example shown in Figure 6, two variables have been defined thus far. "My_var" has been given an index of zero and "Your_var" an index of one.

_0	My_var
_1	Your_var
_2	null

FIGURE 6

Example of Variable List Used During Codification

Procedure Hash Table and Nodes. The organization of the collection of codified procedures is done in a similar manner as the atoms. A hash table is used to gain access to a group of procedures with identical names using a hash of the procedure name. Once again, collisions may occur so the procedure nodes are kept in a linked list. The procedure node has three entries. The first is a pointer to the actual procedure name stored in the atom heap. The second is a link to the next procedure node should there be a collision. The third is a pointer to the codified facsimile of the first of the set of procedures with the name specified in the first field. Since more than one procedure may have the same name, the first entry in the codified facsimile of any procedure is the link to the codified facsimile for the next procedure with the same name. Figure 7 illustrates this. "likes", "hits", and "steals", all hash to the same value. So a procedure node is allocated for each, and the nodes are linked by collision pointers. The group of procedure facsimiles named "likes" make up the second dimension of the data structure.

Not only are nodes made for procedure heads put in the linked list, but for call names also. Then if a procedure is later codified that has the same name as a previously defined call, a procedure node will have already been built. This is why Figure 7 shows no codified facsimile for "rich" and "hits". They result from calls of procedures in the database, and are not procedures themselves. There may be cases where a procedure does not exist with the same name as the earlier defined call. So why even bother defining a procedure node? The major reason is that

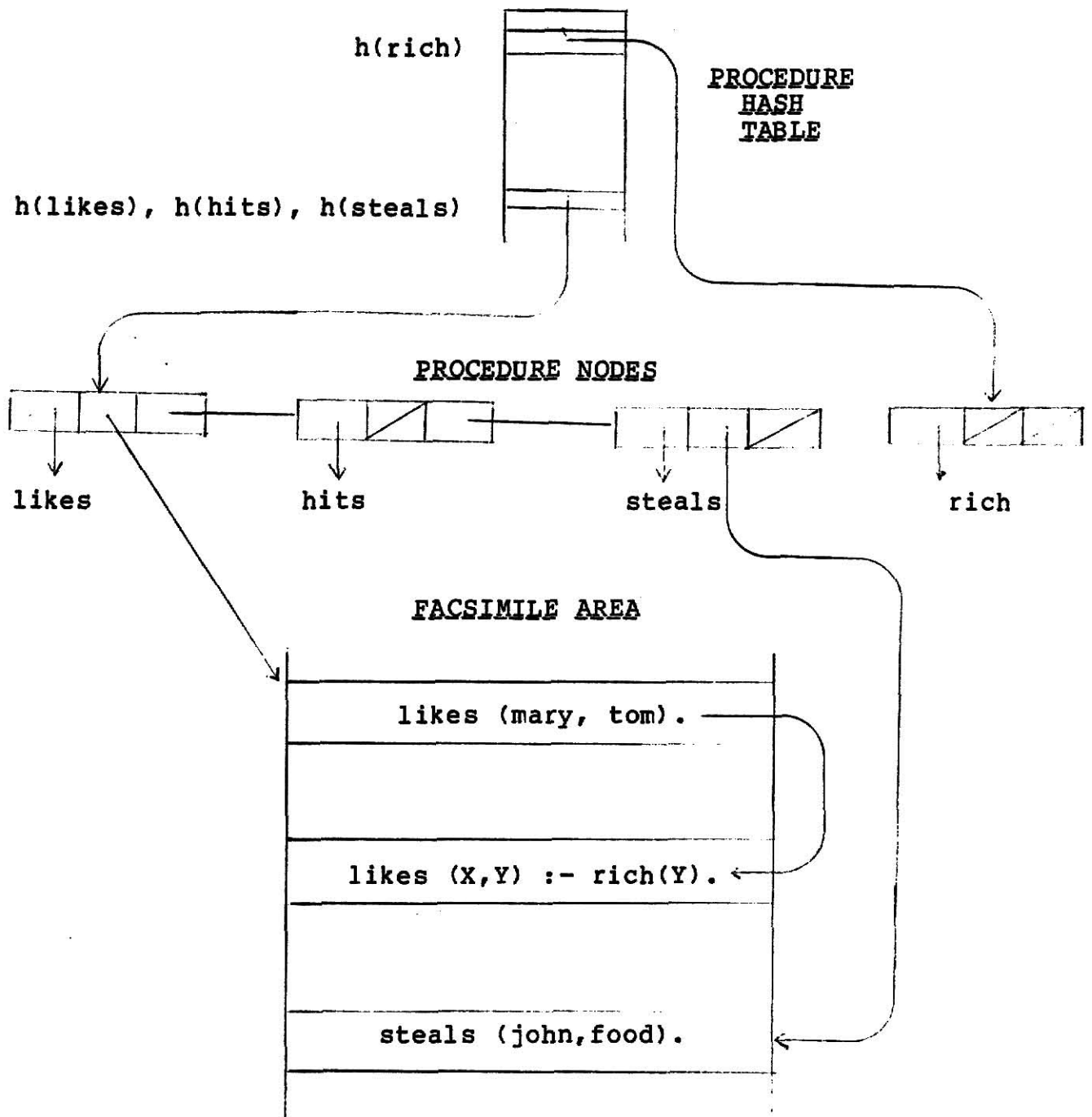


FIGURE 7

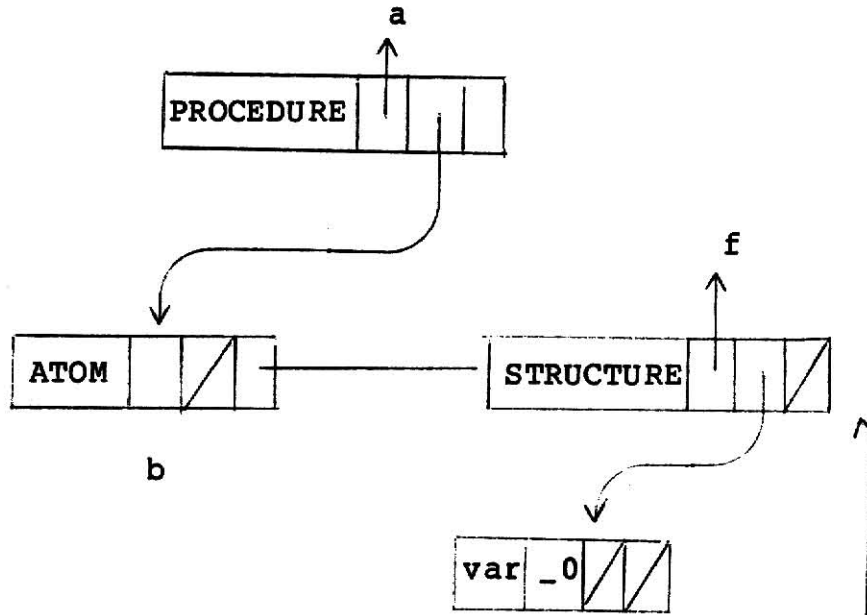
Procedure Hash Table and Node Organization

a procedure can be defined at run time via the built-in predicate called `assert`. It would be attached to the procedure node. Any call entry previously defined with the same name as the new procedure would be pointing to the correct procedure node.

Codification Stack. The last major data structure used by the codification step is a stack that is used for storing information for structures that have not undergone final encoding. The need for the stack arises from the design decision to linearize the binary tree of a procedure to save memory. Parameters that occur at the same level in the tree are put into final codified form one after another. If a parameter happens to be a structure, the skeleton of the structure (organization of its parameters) is defined elsewhere in the facsimile area. So the entry for the structure must contain a pointer, called a skeleton pointer, to where the structure skeleton is fully defined. Two pointers are saved on the stack each time a structure is encountered during linearization of the binary tree. One points to the binary tree node which contains the root of the intermediate form of the structure. The other points to the location in the codified facsimile area allocated to the current parameter, the structure. The second entry is saved so that that location can later be filled in with the skeleton pointer to the actual structure definition. This is illustrated in Figure 8. Here, the second parameter is a structure, "f(x)". The facsimile entry is given a tag denoting it to be a structure. One pointer on the stack points to the facsimile entry. The other pointer points to the binary tree node containing "f(x)".

a(b,f(X))

Binary Tree Form



Facsimile

a(b,f(X))

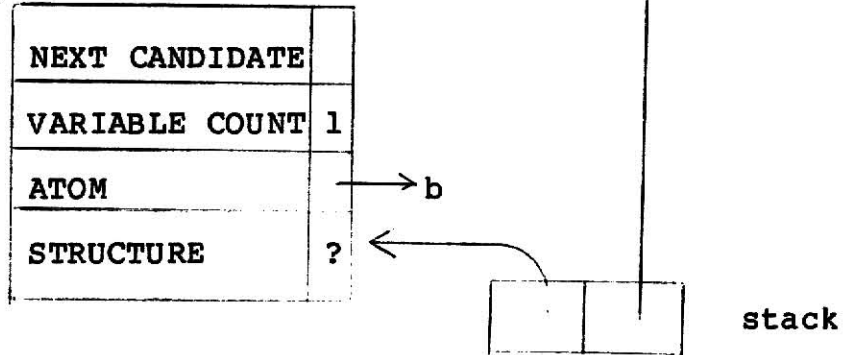


FIGURE 8

Use of Codification Stack

Now that the data structures used by the codification step have been described, it is now possible to explain the actual codification process.

Codification Algorithms

The codification process includes three major steps that are performed on every procedure of the database.

- a. Build binary tree
- b. Linearize tree into final form
- c. Link the final form to the end of the list of procedures with the same name

Build Tree. The build tree function is recursive, which is reasonable since the structure it is building is recursive. The parts of the binary tree in this context have the following meaning. The left subtree corresponds to the first child, that is, a pointer to the first parameter of a particular structure. The right subtree is the sibling of the current parameter.

The main task of this procedure is to determine when it should build a left subtree or a right subtree. The rules are rather simple. When obtaining tokens of the source PROLOG statement, if a left parenthesis is encountered, it is known that the next token will be a parameter of the current one. Thus a left subtree is allocated in anticipation of the upcoming parameter. If the token obtained is a comma, then it is known that the next token will be simply another parameter, thus a sibling (right subtree) node is allocated. The algorithm is outlined in Figure 9.

```

build_tree (tree_node_pointer)
{
    get a token

    if token is an atom
        fill node with atom
        return (build_tree (tree_node_pointer))

    if token is variable
        put variable information in node
        return (build_tree (tree_node_pointer))

    if token is '('
        /* Last atom must have been a structure name or procedure
           name. */
        set tag field of node to denote structure definition
        tree_node_pointer->child_pointer = newly allocated node

        /* Go left to get parameters. */
        build_tree (tree_node_pointer->child_pointer)

        /* Continue right. */
        return (build_tree (tree_node_pointer))

    if token is ','
        /* Another parameter or call is coming up. */
        tree_node_pointer->sibling = newly allocated node

        /* Continue right. */
        return (build_tree (tree_node_pointer))

    if token is ':-'
        /* A call is coming up. */
        set tag of node to indicate call identifier
        tree_node_pointer->sibling = newly allocated node
        return (build_tree (tree_node_pointer->sibling))

    if token is ')'
        /* Done with this level, so go up a level. */
        return (success)

    if token is '.'
        /* Completely done with procedure. */
        return (success)
}

```

FIGURE 9

Algorithm for Building Binary Tree for a Procedure

Link Procedure. The second step of the codification process is to link the facsimile area for the procedure to the end of the linked list of procedures with the same name. This allows for easily obtaining another candidate procedure for unification in case one fails in the unification attempt.

When preparing to linearize the intermediate binary tree form of the procedure, it is known where the final form of the procedure will be put. It is laid down immediately after the previous procedure, in the facsimile area. Figure 10 shows the pertinent data structures. To link the procedure to the end of the linked list of procedures with the particular name, it is necessary to gain access to the head of the linked list. This is done by using the hash value of the procedure name to index the procedure hash table. The hash table entry points to the head procedure node for all procedure names that hash to that value. If one does not exist, it is created.

Once the correct procedure node is found, then access to the second dimension of this structure is gained. This is the linked list of codified procedures with identical procedure names. This list is traversed until the end is reached, signified by a nil link. This link is changed to point to the procedure currently being codified. Since the procedure currently being codified is now the last procedure of the linked list, its link entry, the NEXT CANDIDATE pointer, is set to nil.

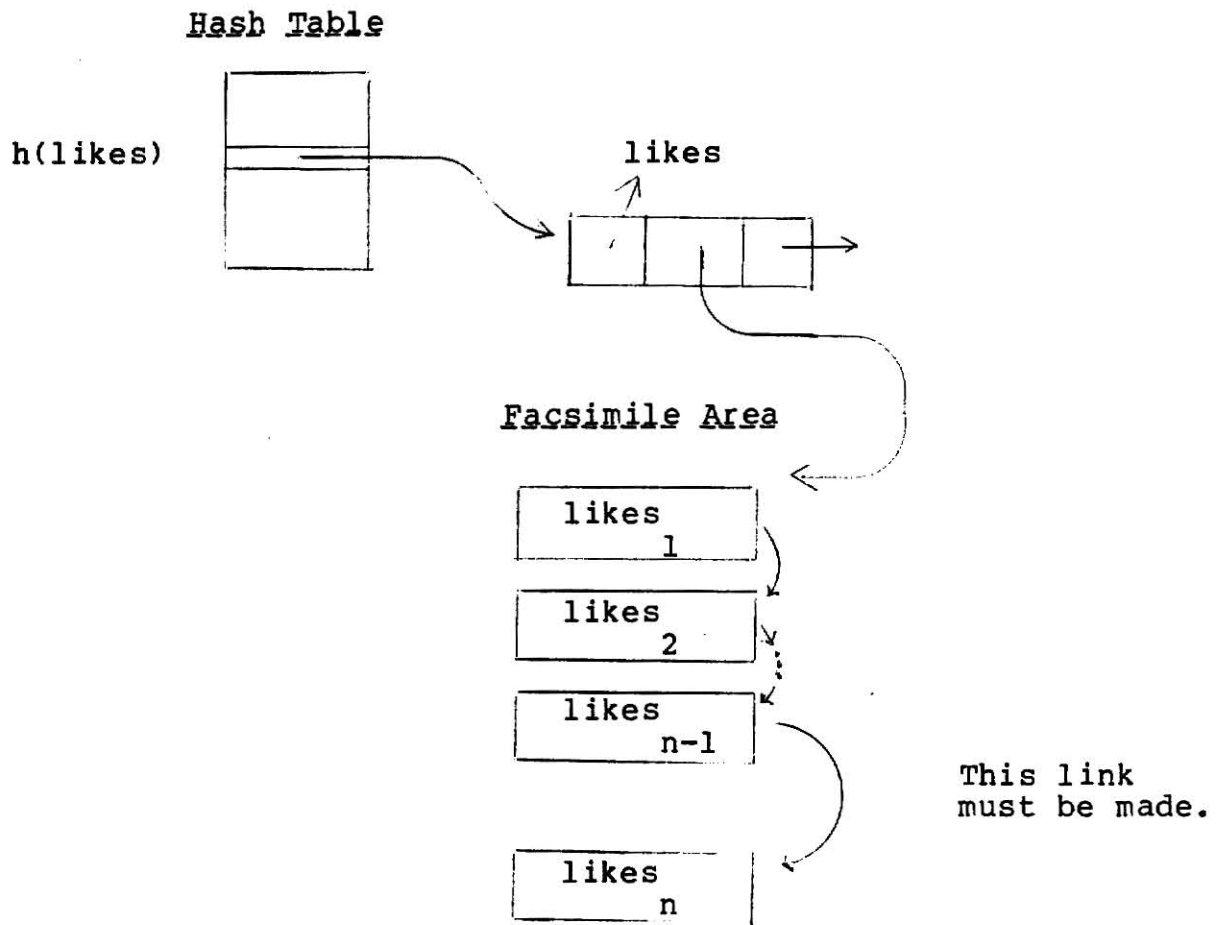


FIGURE 10

Linking a Procedure to the List of Procedures
with Identical Names

Linearizing the Binary Tree. The last major task of the codification function is to put the binary tree in a linear form. This function is somewhat more involved. After the area reserved for the procedure being codified has been linked into the list of procedures with identical names, the codified procedure can be laid down in that reserved space. Two pieces of bookkeeping information are included, followed by the linear form of the information in the binary tree. The important steps of this

function are:

1. Insert bookkeeping information in facsimile area
2. Codify the procedure head
3. Codify the calls
4. Codify any structures stacked in performing the above steps

The first entry in the codified facsimile is the link to the next candidate, which, as mentioned previously, is set to nil. The second entry in the codified procedure is the variable count of the procedure. That is the number of distinct variables in the procedure. A list of the variables is made during codification into the binary tree structure, thus the procedure variable count can be found by counting the entries in this list.

The next entries in the space of the codified procedure are the call and parameter entries. The order of laying down the information is shown.

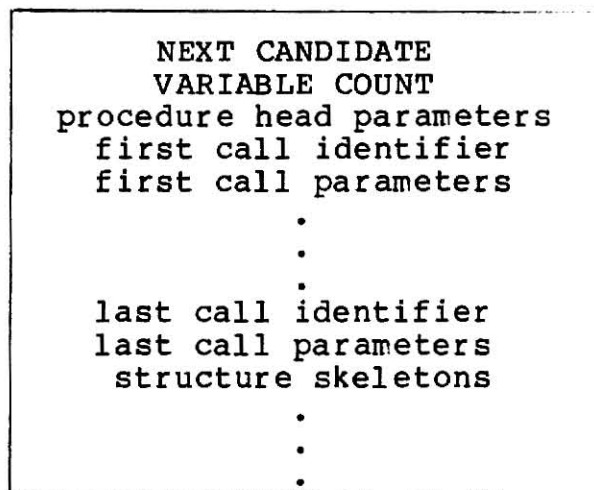


FIGURE 11

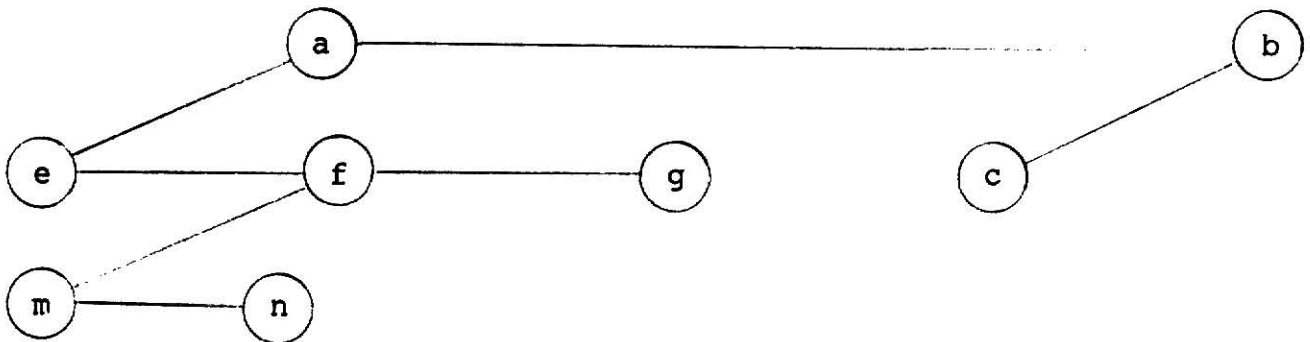
Codified Procedure Format

In order to lay down this information in the correct order, some important principles of binary trees are used. Recall that all nodes of the binary tree connected by the next sibling pointer (right subtree) are considered to be of the same level, whereas a node pointed to by the child pointer (left subtree) defines the first parameter of a sublevel. By the scheme used, the highest level elements (linked by right subtree pointers) are nodes for the procedure and call names. The left subtree pointer of any of these nodes accesses the first parameter of the respective procedure and calls. The parameters of the same tree level are accessed by the right subtree pointer. If any of these parameters of the procedure and call nodes have left subtree pointers, then these parameters are structures. Furthermore, parameters of structures themselves can be structures due to their recursive definition.

The scheme for linearizing the binary tree is to first obtain the first level parameters. This is done by moving down the tree from the first node, the procedure name node, using the left subtree pointer. The procedure head's first level parameters are retrieved by traversing the nodes connected by the next sibling (right subtree) pointers. This concept is shown in Figure 12. If any of these parameters are themselves structures, then it must be remembered to come back later and lay down the structure skeleton after all the first level nodes are processed. This is done by saving the binary tree location on the codification stack along with the location of the entry in the codified facsimile that will have to be filled in with a skeleton pointer.

Procedure: $a(e, f(m, n), g) :- b(c).$

Binary Tree Form



Order of Initial Facsimile Entries

NEXT CANDIDATE
VARIABLE COUNT
entry for "e"
entry for "f"
entry for "g"
.
.
.

FIGURE 12

Obtaining Top Level Parameters of Procedure Head

An example of how the codification stack would look after all of the top level parameters of a fact " $a(b(c), d(e))$ " are codified is shown in Figure 13.

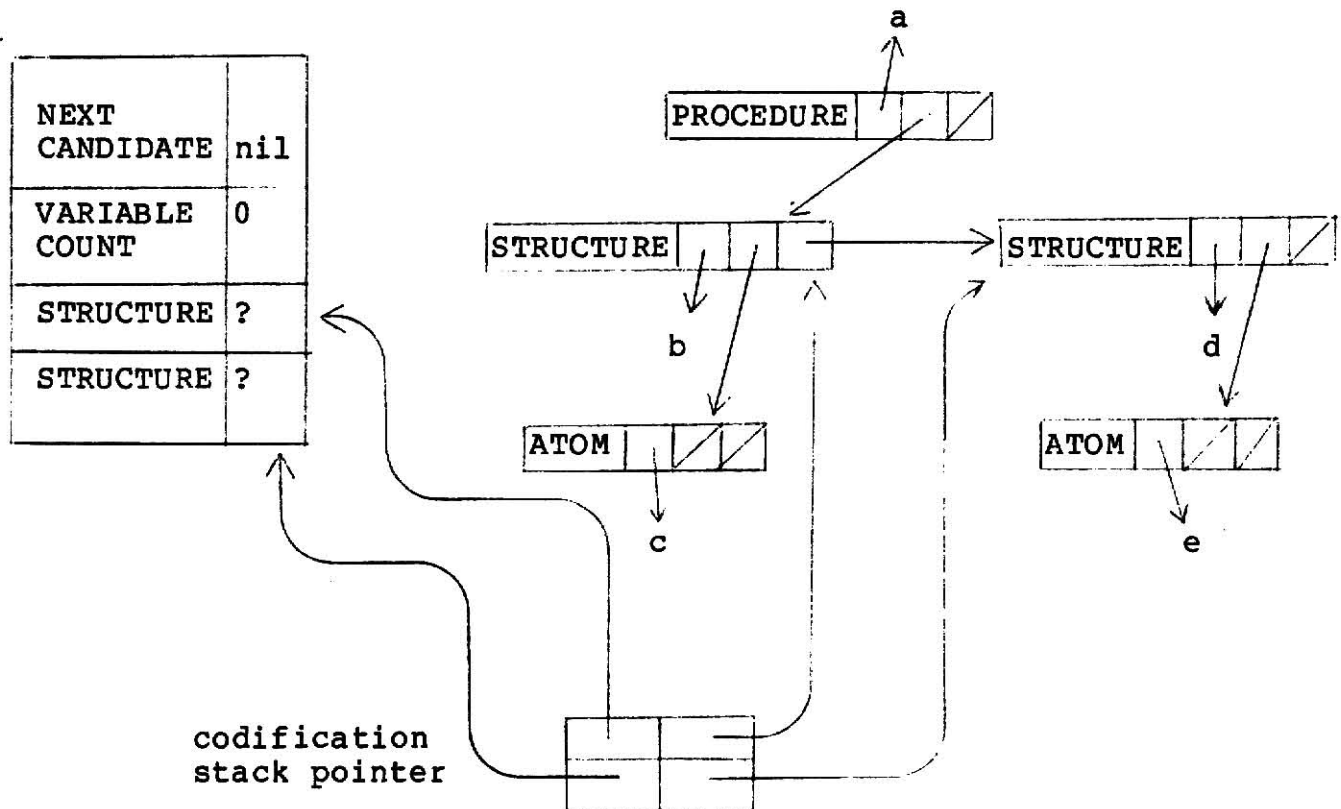


FIGURE 13

Stacking of Structure Information
During Linearization of Binary Tree

After the first level procedure head parameters are put in place, the calls can be processed. The calls are accessed by traversing the right subtree pointers of the highest level in the binary tree. What is put in the codified facsimile that corresponds to the call name is not a pointer to the call name itself. Rather it is a pointer to a procedure node for procedures of the same name as the call. This pointer is referred to as the call identifier. As mentioned before, the reason for using a pointer to the procedure node is that this provides quick access to

potentially matching procedures when unification is attempted for the call. Figure 14 illustrates the use of the call identifier.

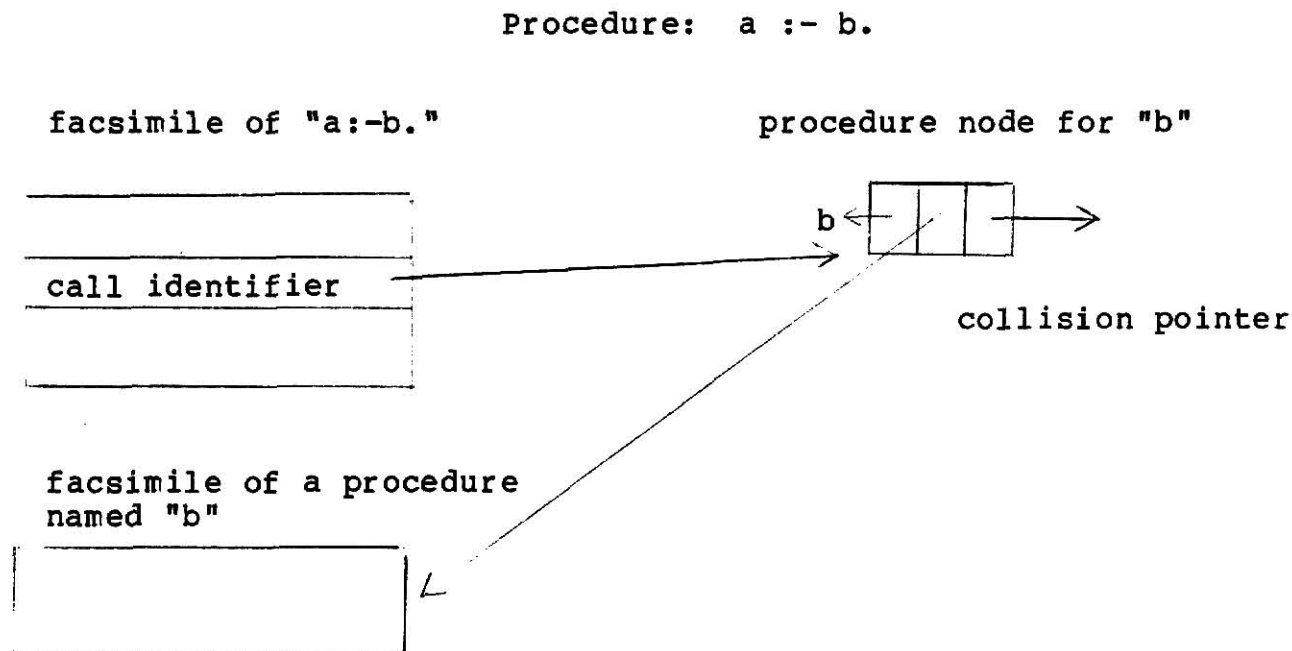


FIGURE 14

Use of Call Identifier

After the call identifier is put in place, the call's first level parameters are put in place, just as the procedure head's parameters were. Stacking of structure information is done as before. This process is repeated for each call.

After a particular level of a tree branch is codified, the stack is popped so that codification of a saved branch can start. A pointer to the facsimile entry where the skeleton pointer for the structure branch just popped is to be can be filled in. Its value is the address of the next available entry in the facsimile where the structure branch will now be codified. The pointer to the branch of the tree holding the structure is the other entry

popped from the stack. The flowcharts of Figure 15 and 16 describe the linearization of the binary tree. The latter is the flowchart for a subroutine used by the first.

Once all the calls and their parameters are in place, the stacked structures are put in the codified facsimile area next. A similar scheme is used as before with only parameters of the same level in the binary tree being put in sequence in the facsimile, with structures information being stacked. Codification is complete when all stacked structures have been processed.

After a procedure's binary tree form has been linearized into its final form, its nodes are deallocated with a recursive procedure.

This entire process of putting a procedure into an intermediate binary tree form and then linearizing the tree is repeated for all procedures in the database. Once the entire data base has been codified, the interpreter can go to work obtaining and solving queries to the data base.

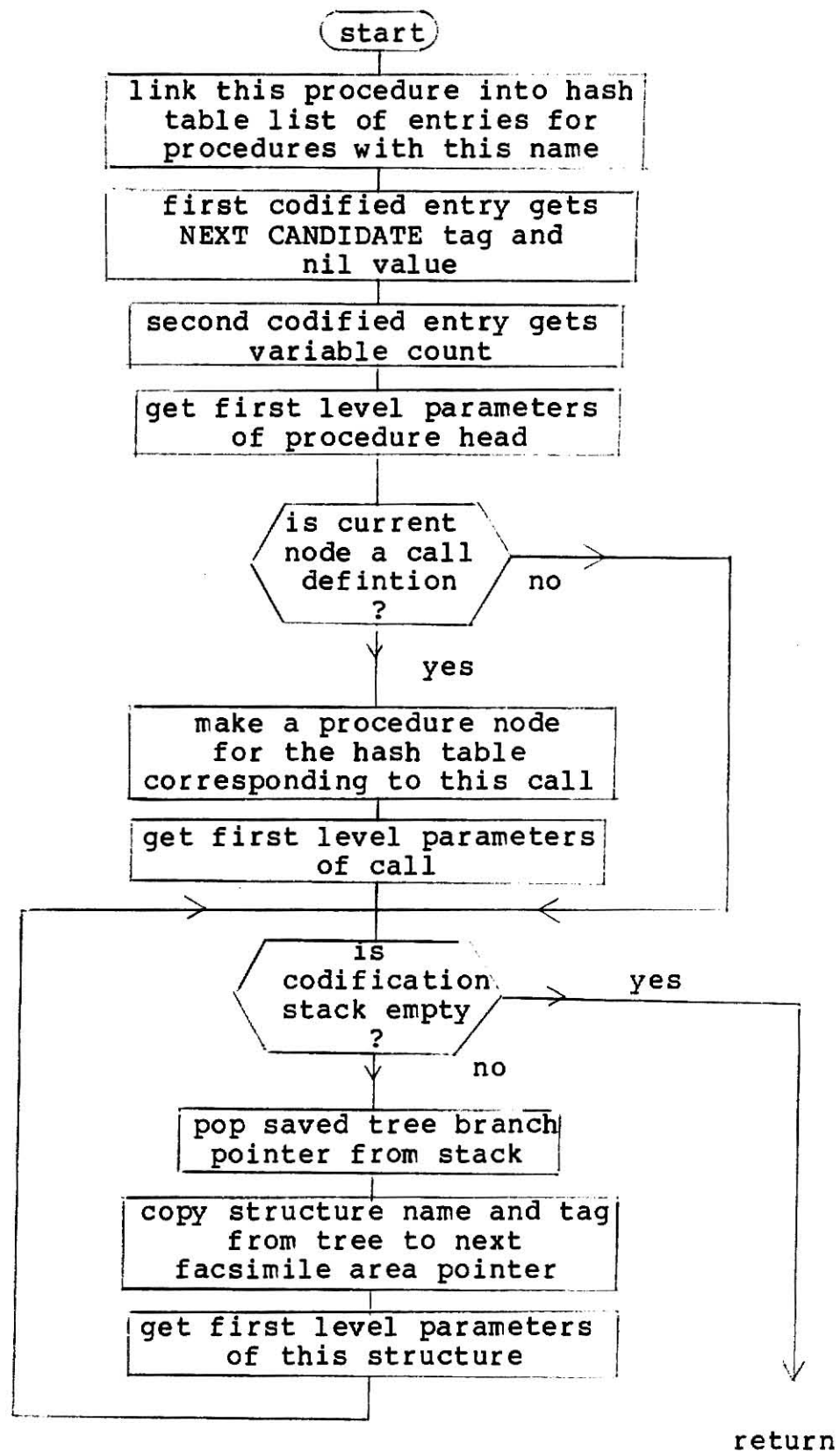


FIGURE 15

Linearize Tree Flowchart

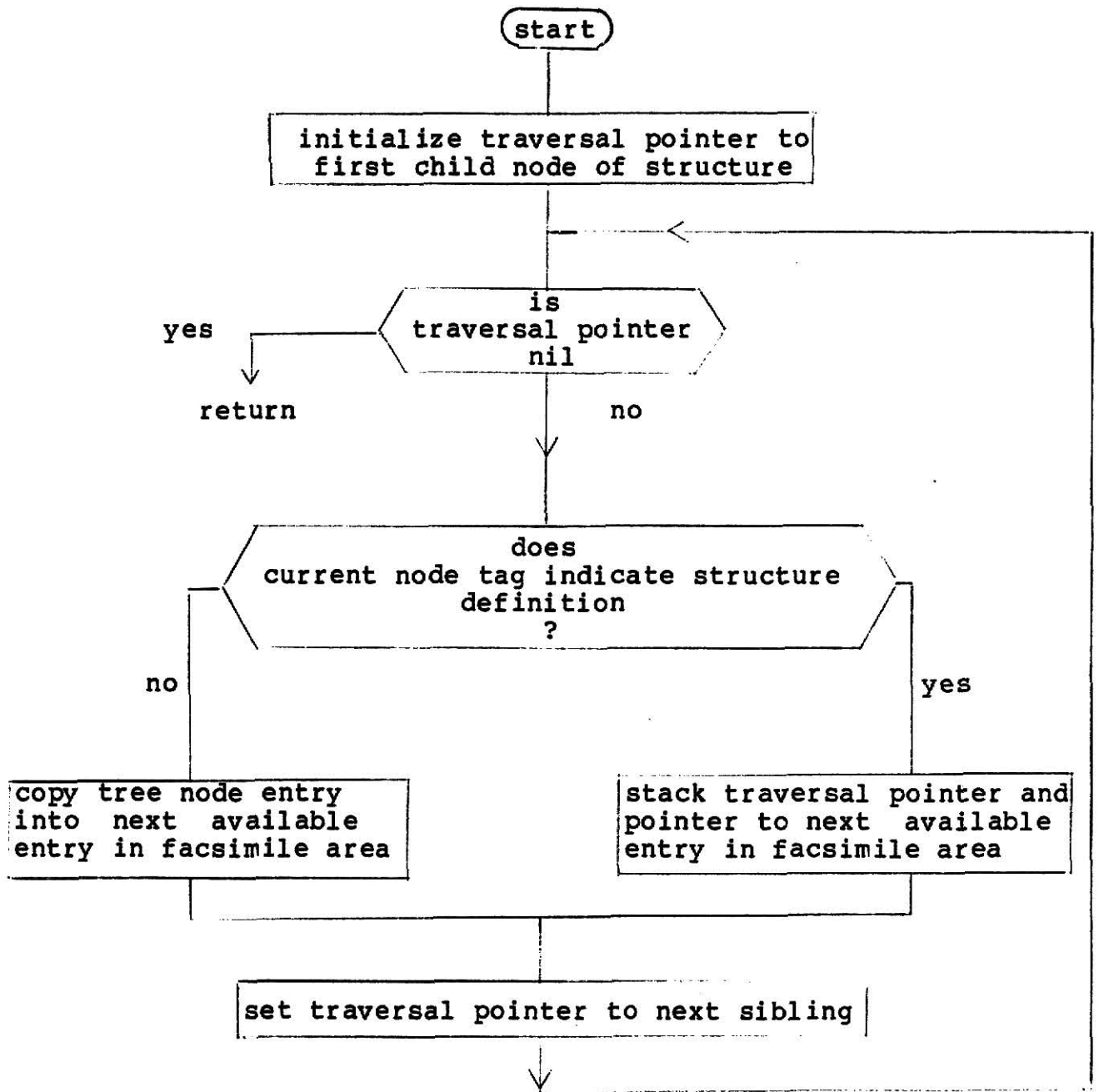


FIGURE 16

Get First Level Parameters Flowchart

V. INITIALIZATION

The initialization routine obtains the user's query and initializes the stacks and registers. The flow diagram for the initialization step is shown in Figure 17.

The input goal is codified in the same manner as procedures except the procedure name is nil. A pointer is saved to the point in the codified facsimile space where the goal begins so that the memory can be reclaimed after the attempt to solve the goal. As part of the initialization process, the CURRENT PROCEDURE pointer is set to the location of the main goal's coded form.

The MOST RECENT PARENT pointer keeps track of which frame was created for a procedure whose call the interpreter is currently trying to solve. Recall its purpose is to identify the frame that indicates which call is to be solved next. That call is pointed to by the RETURN pointer of the parent frame. Each frame has at a minimum a PARENT FRAME pointer and a RETURN pointer. The initialization routine initializes these two entries to nil for the input goal frame since it has no parent, and thus no return call. The trail stack is initialized as empty since no bindings have been made to any variables. Space is allocated on the frame stack for any main goal variables.

Once the initializations take place, the interpreter can

determine the first call to be solved. This is done by the call selection routine, and will be described in the next chapter.

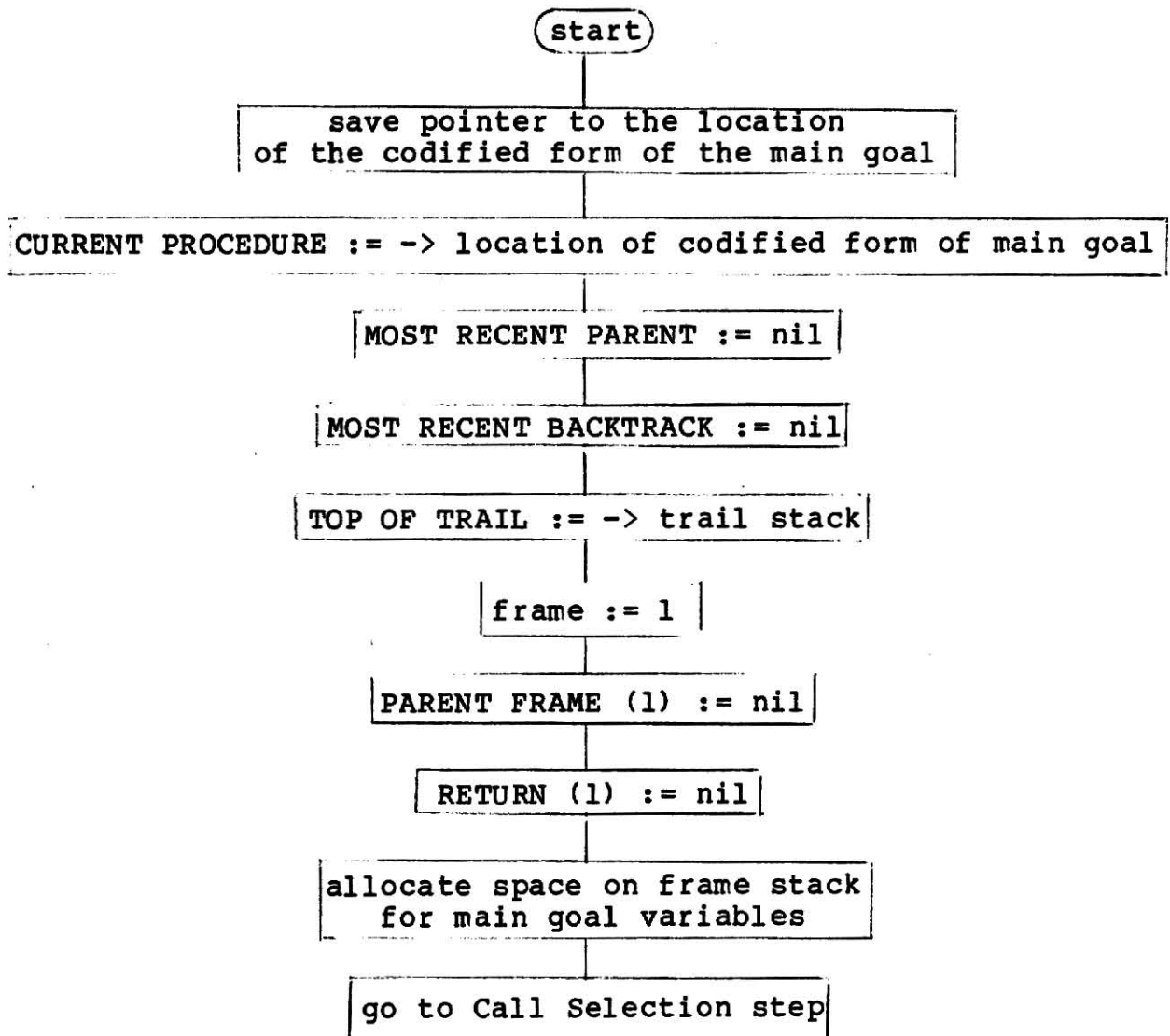


FIGURE 17
Initialization Flowchart

VI. CALL SELECTION

The call selection step of the interpreter determines the next call to be solved. There are two ways control passes to this portion of the algorithm. The most common way is after the frame creation step, in which a frame is created on the stack for a procedure just successfully unified. The other way is from the initialization step in which a frame is created for the main goal. The latter case is treated no differently from the first as far as the call selection routine is concerned.

The call selected is the first call of the just successfully unified procedure if that procedure is not an assertion. For example, if the procedure just unified is "a :- b, c, d.", then "b" is the call selected to be tried in the next unification attempt.

If the procedure just selected is an assertion, then it has no calls to solve. First the interpreter looks at the RETURN pointer just put into the frame by the frame creation step. If that pointer is null, then the interpreter looks to see if the frame created for the parent procedure, indicated by the MOST RECENT PARENT, has any calls to solve by looking at its RETURN pointer. If the RETURN pointer indicates that no more unsolved calls are in the parent frame, then the RETURN pointer of the grandparent frame is examined through the use of the PARENT FRAME

pointer located in the frame pointed to by the MOST RECENT PARENT. The search continues until an unsolved call is found, or until the MOST RECENT PARENT frame pointer is nil. The algorithm is described by the flow diagram of Figure 18.

Once a call is found, it is determined if the call has the same name as any procedure in the data base. This is determined by looking at the CALL IDENTIFIER, pointed to by CURRENT CALL. The CALL IDENTIFIER points to the head of a linked list of procedures that have an identical name. This is shown in Figure 19.

At first glance, it might appear advantageous to take out a level of indirection and have the CALL IDENTIFIER point directly to the facsimile of the first matching procedure. The reason this is not done, as mentioned earlier, is because if there initially are no matching procedures, the pointer value is nil. But if a matching procedure is later added via the assert predicate (allows for a dynamic database), no access to it will be possible through CURRENT CALL.

Assuming a list of candidate procedures has been found, it must be determined if any of the candidate procedures will unify with the call pointed to by CURRENT CALL. Unification requires that the parameters of the call and the candidate procedure to match, and is described in the next chapter.

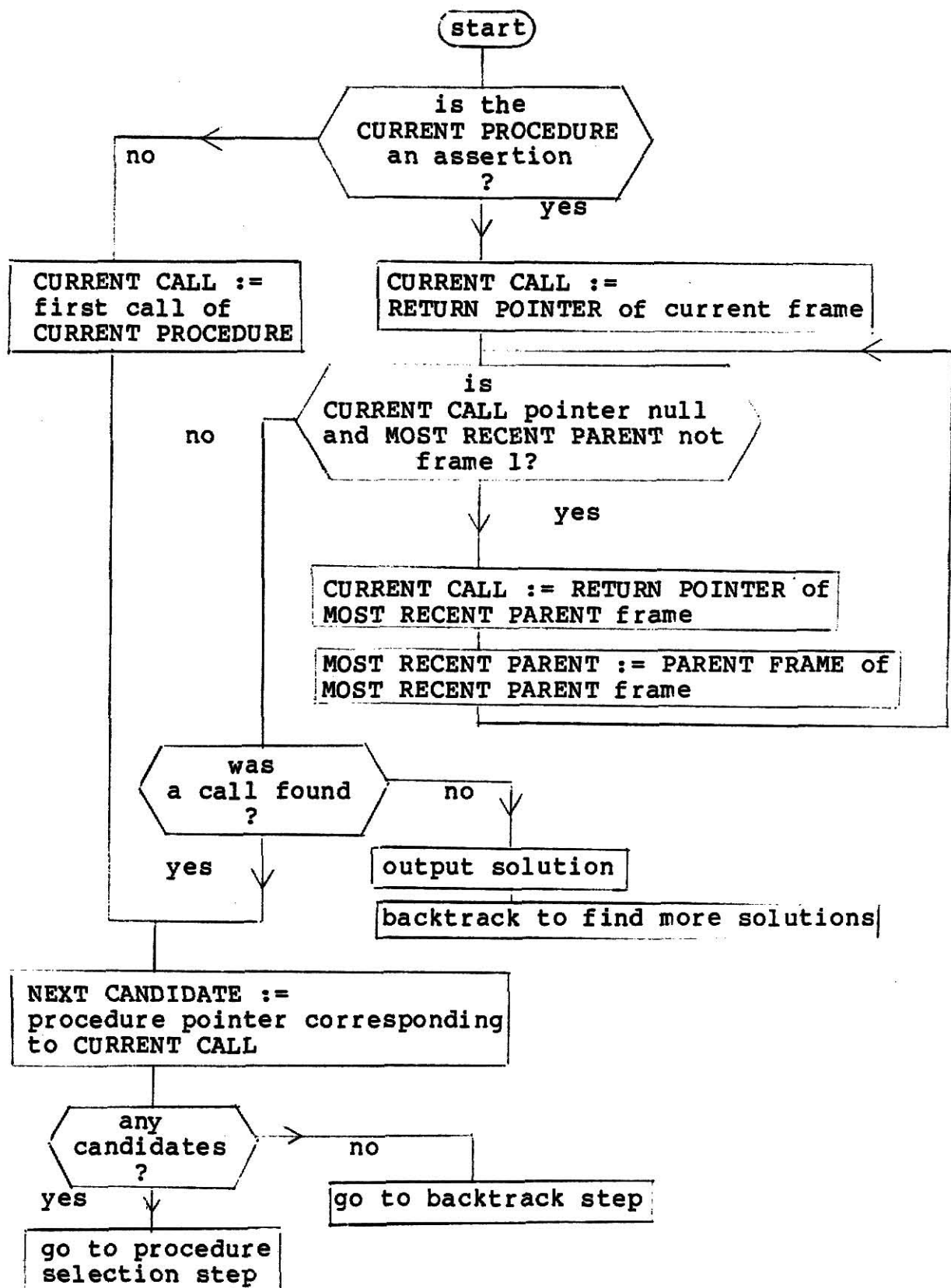


FIGURE 18
Call Selection Flowchart

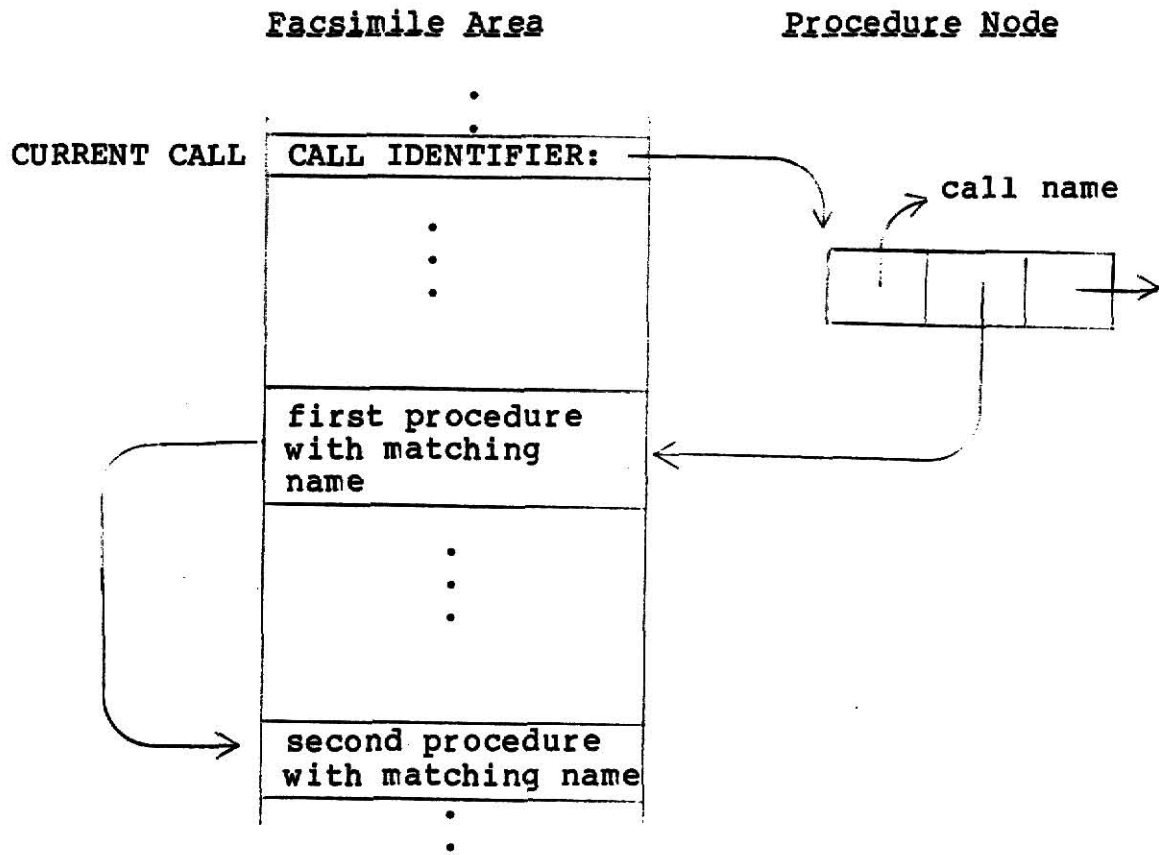


FIGURE 19
Accessing Procedures Matching Call Name

VII. PROCEDURE SELECTION

This routine attempts unification between the current call and the procedure whose name matches until a success is found, or until it runs out of candidates. Also, any variable bindings necessary during unification are made.

Use is made of the space beyond the current frame where a new frame will be created when a successful unification occurs. Some of the data that is stored in a new frame, especially variable bindings, is determined during the unification attempt. If the unification is successful, then an advantage is that less information needs to be filled in during the frame creation step. If an unsuccessful unification occurs, no harm is done because the invalid data will eventually get written over.

The procedure selection routine calls a recursive function called Unify to do the actual unification attempt. It is naturally recursive, considering it is operating on the recursive structures of the candidate procedure and the current call. The flowchart for the procedure selection step is shown in Figure 20. Prior to calling the recursive routine, the algorithm saves TRAIL POINTER and CURRENT CALL in the next frame to be created. This is because their contents will be modified during a unification attempt and will have to be reset for a new attempt. As mentioned before, the pointers have to be saved in the new frame anyway if it is a backtrack frame (Figure 21).

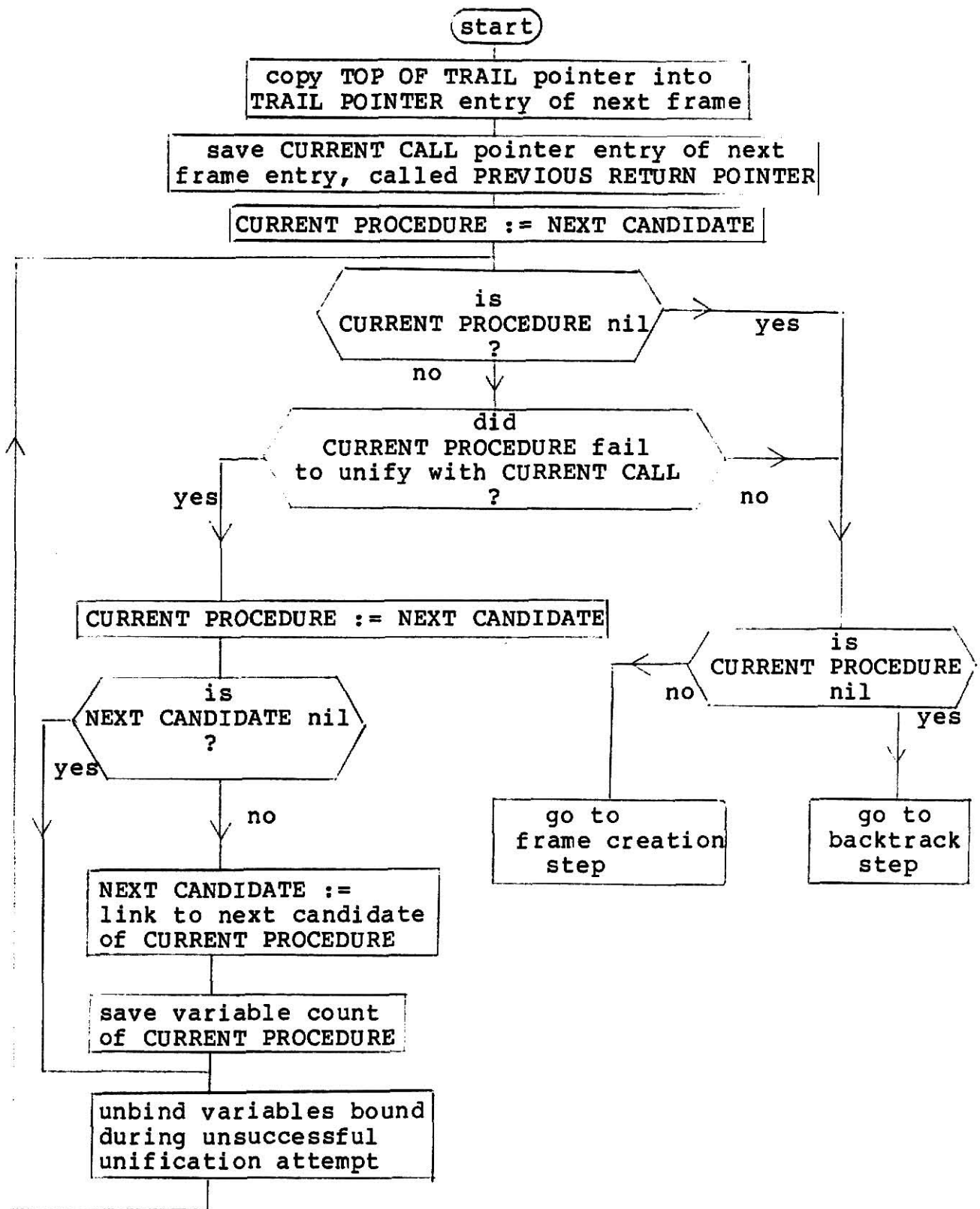


FIGURE 20
Procedure Selection Flowchart

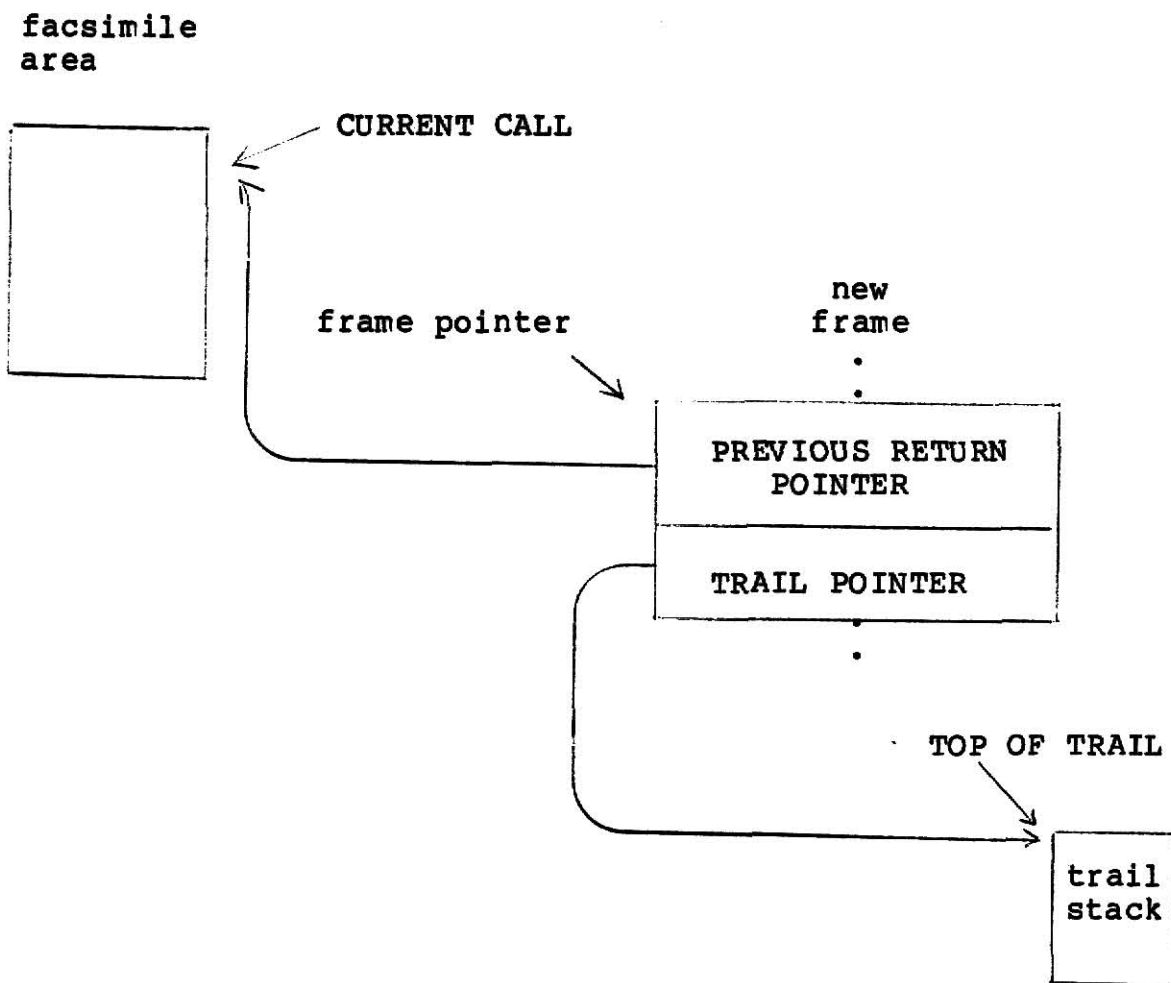


FIGURE 21

Recording Pointers in New Frame During Unification

Unification

Control is passed to the unification routine once a procedure has been found that matches the current call by name. More restrictive filtering could be done here by also assuring that the number of parameters also matches.

Unify is a recursive routine because it operates on a recursive data structure, the tree. Even though it is put in a linear form for memory savings, conceptually it is still a tree. Unification is performed by attempting to match each parameter of the current call with the corresponding parameter of the current procedure. The search algorithm is a commonly used preorder binary tree traversal. The algorithm is presented by Kruse and is given below (8:195).

```
procedure preorder (root: pointer);
begin
    if root <> nil then
        begin
            visit (root)
            preorder (root->right)
            preorder (root->left)
        end
    end
```

FIGURE 22

Binary Tree Traversal Routine

The recursive algorithm first checks to see if the root nodes are equal. It then calls itself to see if the right nodes are equal. What was the right node becomes the root in the new call.

When the routine has gone right down the tree as far as it can, it checks to see if the left nodes are equal by a recursive

call. Figure 23 shows the order the nodes of a tree are searched.

$a(b, c(d, e), f(g)).$

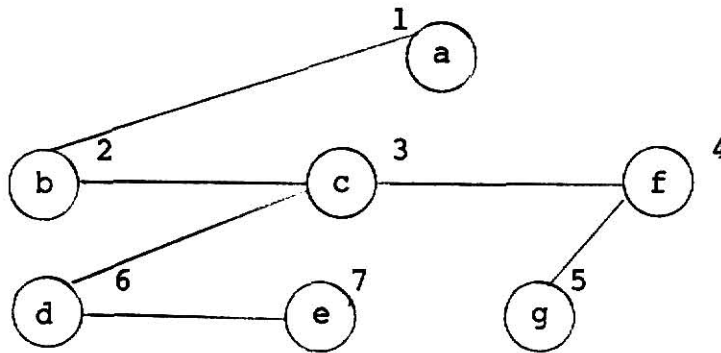


FIGURE 23

Order of Tree Traversal

In order for atom nodes to match, their string pointers simply have to be checked for equality since there is only one copy of a string in the heap. The unification process is complicated because of the possibility of parameters that are variables. Variables that have not been bound to a value can be bound to structures, atoms, or even to other variables (bound or unbound). This means that if a variable is matched with a structure, the nodes for the structure and all of its children do not even have to be traversed. In this case, unification is simpler. But if the variable does previously have a value bound to it, its value must be matched to the corresponding parameters of the other tree. An example is shown in Figure 24.

$a(b, c(d, e), f(g))$ matched with $a(b, X, Y)$

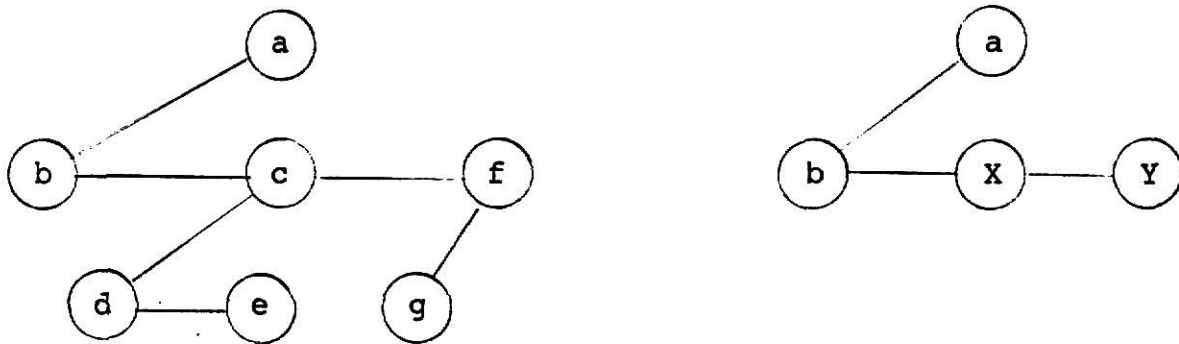


FIGURE 24

Matching Variables with Structures

Variable "X" gets bound to "c(d,e)" and "Y" gets bound to "f(g)". Three nodes do not have to be visited in this case.

Parameters can be one of three types; variables, atoms, or structures. Thus, there are nine possible ways to try to match parameters. The rules are presented below in Figure 25.

In more precise terms, the unification process is started with the CURRENT CALL pointer pointing to the initial entry of the call facsimile, the pointer to the linked list of candidate procedures. The CURRENT PROCEDURE pointer points to the first entry in the codified facsimile of the procedure, its variable count. The first parameter of the respective trees are the next entries in the respective facsimiles. The unify procedure starts by incrementing each pointer to make them point to the first parameter.

PARAMETER	PARAMETER	RULE
atom	atom	Matches if pointers are equal.
atom	variable	Matches if variable is unbound or bound to atom with same atom 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 rules of 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 of this table.

FIGURE 25

Parameter Matching Rules

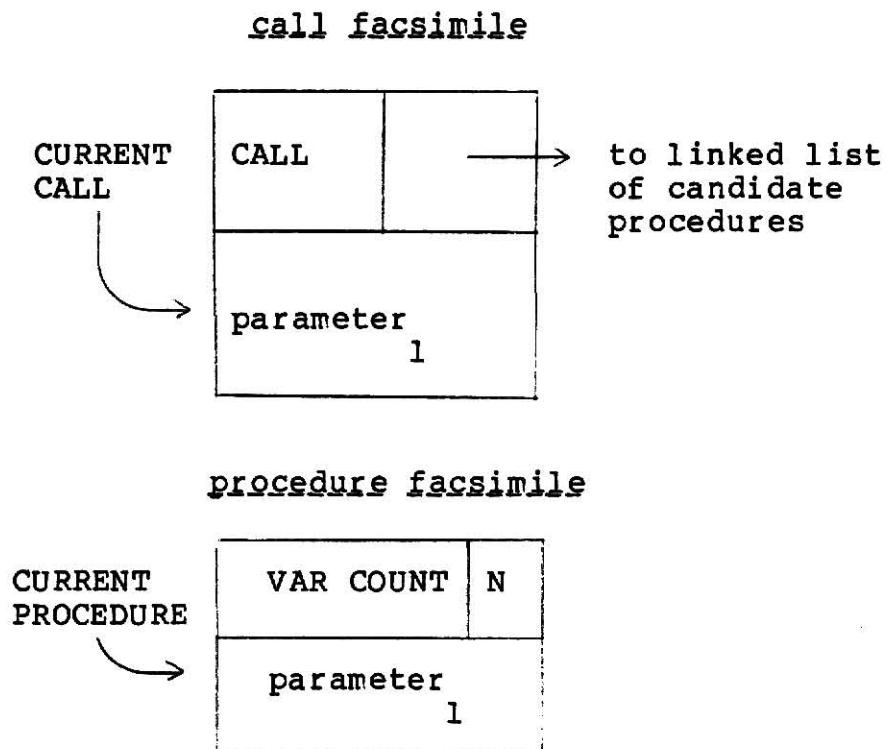


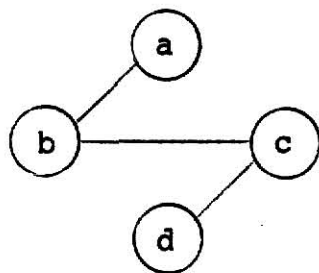
FIGURE 26

Initializing Pointers for Unification Process

As parameters are matched, the CURRENT CALL and CURRENT PROCEDURE pointers are normally each incremented to allow comparison of successive parameters. The simplicity is destroyed when a structure is encountered. This corresponds to the necessity to search left in the binary tree analogy. This is achieved by conceptually following the left link, known as the skeleton pointer, to the facsimile entries that defines the structure. Once there, the recursive traversal continues with the usual sequential matching of the respective parts until the encountering of another structure requiring yet another skeleton pointer indirection. The indirection is illustrated in Figure 27.

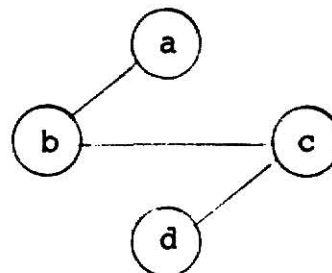
Current Call

a(b, c(d))

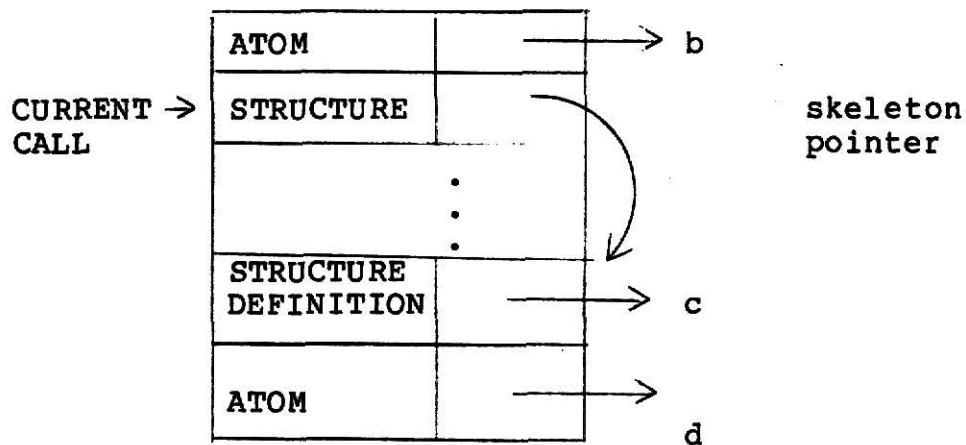


Current Procedure

a(b, c(d))



call facsimile



procedure facsimile

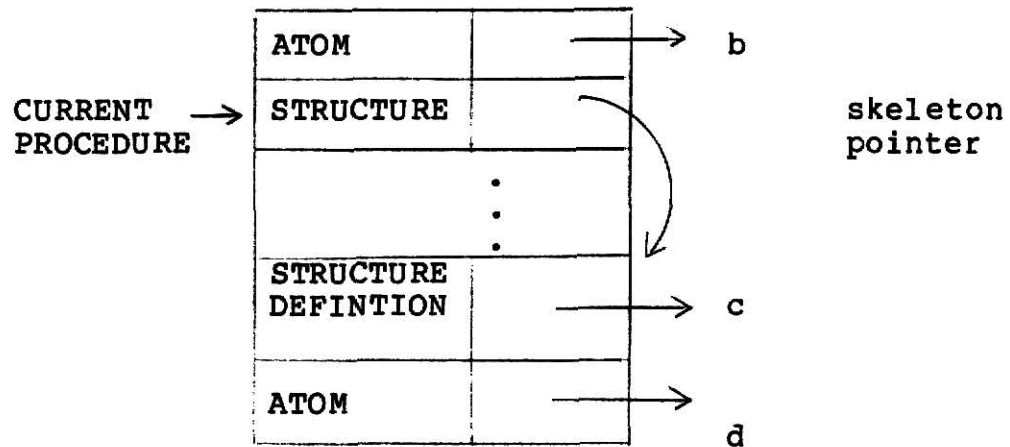


FIGURE 27

Skeleton Pointer Indirection

As stated earlier, if an attempt is made to match a variable with a structure, the left traversal in the binary tree, that is, the skeleton pointer indirection, does not take place.

Variable Binding

When variables of a procedure are encountered, space is allotted in the new frame of the frame stack for it. Three pieces of information are stored for each variable on the stack. The first is a tag that indicates whether the variable is unbound, bound to a variable, bound to an atom, or bound to a structure. The second piece of information is a pointer to the variable's value. Its value may be a binding to an atom, another variable, or a structure. When it is bound to a structure, the skeleton of the structure is specified by the codified facsimile of it. Thus, the pointer would point to a location in the facsimile area. The structure itself may contain variables whose values are defined in another frame of the stack. This creates the necessity for the third piece of information associated with each variable entry of the stack, the environment pointer. It indicates the frame in which any variables of the structure are defined in the frame stack.

Since the scenario is quite involved, it is best described with an example. This example will also illustrate the previous unification concepts. The example has a database of four rules and one main goal. The rules and their codified facsimile are shown in Figure 28.

Original Database

```

a(M, N) :- b(N, M).
b(c(X), d(Y)) :- e(Y), f(X).
e(g).
f(h).
:- a(I, J).

```

Codified Facsimile

```

a(M, N)
:- b(N, M).

```

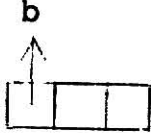
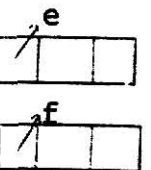

NEXT CANDIDATE	/	
VARIABLE COUNT	2	
VARIABLE	-0	
VARIABLE	-1	
CALL		→
VARIABLE	-1	
VARIABLE	-0	
		
NEXT CANDIDATE	/	
VARIABLE COUNT	2	
STRUCTURE		
STRUCTURE		
CALL		→
VARIABLE	-1	
CALL		→
VARIABLE	-0	
STRUCTURE SKELETON		→ d
VARIABLE	-0	
STRUCTURE SKELETON		→ c
VARIABLE	-1	
		
NEXT CANDIDATE	/	
VARIABLE COUNT	0	
ATOM		→ g
NEXT CANDIDATE	/	
VARIABLE COUNT	0	
ATOM		→ h
GOAL	/	
VARIABLE COUNT	2	
CALL		→
VARIABLE	-0	
VARIABLE	-1	
		

FIGURE 28

Database for Unification Examples

The first frame built for the main goal, `" :- a(I, J) "`, has variable entries allocated in the frame stack for `"I"` and `"J"`. They are initially unbound. The environment pointers have no meaning since the variables are not bound to structures. Their value pointers are nil. The PARENT FRAME and RETURN POINTER will be ignored in this example.

frame 1 for :- a(I,J).	parent frame		
	return pointer		
	UNBOUND	/	/
	UNBOUND	/	/

FIGURE 29

Frame for Main Goal

The procedure `"a(M,N) :- b(N,M)"` is selected to be the current procedure. Unification will be attempted with it and the current call, `"a(I,J)"`, the only call in the main goal. The parameters of the current procedure, `"M"` and `"N"`, match with `"I"` and `"J"`, respectively. This is because unbound variables automatically match. The value pointers for `"M"` and `"N"` then point to the frame entries for `"I"` and `"J"`.

frame 1 for :- a(I,J).	parent frame		
	return pointer		
	UNBOUND	/	/
	UNBOUND	/	/
frame 2 for a(M,N) :- b(N,M).	parent frame		
	return pointer		
	BOUND TO VARIABLE	/	
	BOUND TO VARIABLE	/	

FIGURE 30

Frame Stack After One Unification

The current call now becomes "b(N,M)", and "b(c(X),d(Y)) :- e(Y), f(X)" becomes the current procedure. Now the parameter by parameter unification provides an interesting result. The structure "c(X)" is matched with variable "N", which is in turn bound to variable "J". Thus variable "J" can be assigned "c(X)". This is done by making the tag for "J" denote that it is bound to a structure. Its skeleton pointer (value pointer) points to the facsimile for "c(X)". Note that the structure "c(X)" contain a variable "X". "X" gets allotted a frame entry in frame three because it is the frame created for the current procedure, which contains "X". Thus variable "J" has an environment pointer indicating that frame three defines the values for the variable in the structure "c(X)". A similar situation occurs for "d(Y)" being unified with "M". which is in turn bound to "I".

frame 1 for :- a(I,J).	parent frame return pointer BOUND TO STRUCTURE 3 BOUND TO STRUCTURE 3	3	3	<p>d(Y) c(X) skeleton pointers</p>
frame 2 for a(M,N) :- b(N,M).	parent frame return pointer BOUND TO VARIABLE / BOUND TO VARIABLE /	/	/	
frame 3 for b(c(X),d(Y)) :- e(Y),f(X).	parent frame return pointer UNBOUND UNBOUND	/	/	

FIGURE 31

Variables Bound to Structures

The current call now becomes "e(Y)". The procedure "e(g)" becomes the current procedure. The atom "g" is matched with variable "Y" of frame three. Variable "Y" gets a tag indicating it is bound to an atom. Its value pointer points to the heap representation of "g", and its environment pointer is undefined since the variable is not bound to a structure.

The current call now becomes "f(X)" and the current procedure becomes "f(h)". The binding of atom "h" to variable "X" can be made. The final frame structure is shown in Figure 32. Note the values of "I" and "J" have the values "d(g)" and "c(h)", respectively.

The Trail

For every binding of a variable of the frame stack, an entry is made in the trail stack. This is done so that if backtracking occurs due to a dead end (no successful unification of a procedure and a call), variables in earlier frames can be reset. This is necessary because values acquired along the way to a dead end are invalid. The value of the trail stack pointer is saved in a backtrack frame. A backtrack frame is a stack frame built for a procedure-call unification that has other untried candidate procedures. That way, when backtracking occurs, the old trail pointer can be retrieved and all variables pointed to from the top of the trail stack to the saved trail stack pointer are reset. Since bindings are made before the new frame is created, the top of the trail must be saved in the location of the yet-to-be-created frame. Figure 33 introduces an example that shows the use of the trail stack.

frame 1 for :- a(I,J).	parent frame return pointer BOUND TO STRUCTURE 3 BOUND TO STRUCTURE 3	3 3			
frame 2 for a(M,N) :- b(N,M).	parent frame return pointer BOUND TO VARIABLE BOUND TO VARIABLE	/	/		
frame 3 for b(c(X),d(Y)) :- e(Y),f(X).	parent frame return pointer BOUND TO ATOM BOUND TO ATOM	/	/		
frame 4 for e(g).	parent frame return pointer				
frame 5 for f(h).	parent frame return pointer				

d(Y)
 c(X)
 skeleton pointers

h
 g

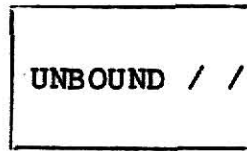
FIGURE 32

Frame Stack after all Variables are Instantiated

Data Base

```
:- a(B).
a(c) :- d(e).
d(e) :- f.
a(g).
```

frame 1
for :- a(B).



← variable entry for B

TOP OF TRAIL

trail stack

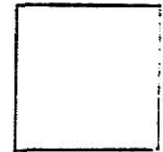
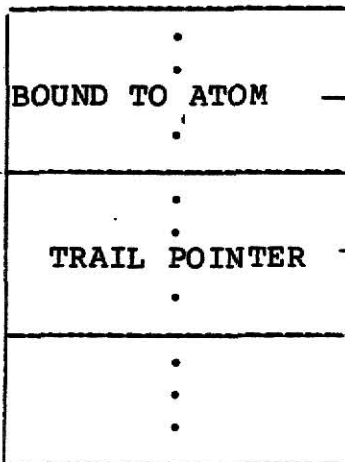


FIGURE 33

Use of Trail Stack

Initially the trail stack is empty. The current call, "a(B)", is matched with "a(c)". However, an untried candidate exists, "a(g)", so frame two is a backtrack point. The previous top of trail is saved in a backtrack frame. Now "d(e)" is the current call. It matches with "d(e) :- f", so a frame is created for it.

frame 1
for :- a(B)



frame 2 for
a(c) :- d(e).

frame 3 for
d(e) :- f

TOP OF TRAIL

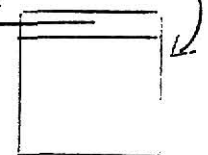


FIGURE 34

State of Frame Stack Prior to Backtracking

At this point, "f" becomes the current call, but no procedures match. Backtracking must occur. Variable "B" was bound to the atom "c" in a frame created along the way to the dead end. Thus "B" is reset, frames two and three are discarded, and the top of the trail is set equal to the value saved in frame two. Now the trail stack is empty once again. The example ends with "g" being bound to "B".

The frame not only contains variable bindings and saved trail pointers, but also some other bookkeeping information. The rest of the frame information is filled in by the frame creation step which will be described next.

VII. FRAME CREATION

The frame stack serves two main purposes. First it records variable assignments. Secondly, it records information needed by the interpreter to determine the next call to be solved.

A frame is created on the stack every time a call is unified with a procedure head. Any variable contained in the procedure has space allocated for it in the frame. Each frame also has associated with it, as a minimum, a RETURN pointer and a PARENT FRAME pointer.

The RETURN pointer points to the facsimile of the next call to be solved. It is normally the call after the current call. If the current call is the last one of a procedure, then the RETURN pointer is set to nil. The call selection routine described in Chapter VI uses the RETURN pointer to determine the next call to be solved. If the call selection routine encounters a nil RETURN pointer described above, then the RETURN pointer of the parent frames are examined until a valid RETURN pointer is found. This implies the necessity to record what frame the parent frame is, so this is the motivation for the second pointer of a frame. The PARENT FRAME pointer points to the frame created for the procedure in which the current call is found.

The possiblility of the need to backtrack in the solution search is evident from the following scenario. It may be the

case that a matching procedure cannot be found for the current call. It may also be the case that earlier in the inference process, a particular procedure was selected for unification when there may have been other untried candidate procedures that might also have unified. Finally, it may be the case that had the interpreter selected one of those other candidate procedures earlier, it might not have hit a dead end later. Provisions have been made in the interpreter to take care of just this scenario. When a dead end is hit, the interpreter backtracks to the last step that had an untried candidate procedure. How the backtracking occurs will be discussed in the next chapter. But some information has to be saved in the frame during the frame creation step in order to allow backtracking to take place later.

Four additional pointers are saved in a frame when other candidate procedures exist for the current call. They are called PREVIOUS RETURN, NEXT CANDIDATE, PREVIOUS BACKTRACK, and TRAIL POINTER.

The PREVIOUS RETURN pointer is set equal to CURRENT CALL. Thus when backtracking occurs to this frame, the CURRENT CALL pointer is reset to the value of the PREVIOUS RETURN pointer to partially reestablish the state the interpreter was in.

The NEXT CANDIDATE pointer is set equal to the location in the codified facsimile of the next untried procedure that has the same name as the current call. Thus when backtracking occurs, the CURRENT PROCEDURE pointer is set equal to the NEXT CANDIDATE pointer.

The PREVIOUS BACKTRACK pointer is a record of the backtrack frame that existed before the current one. This is necessary so

that backtracking can occur to backtrack points even further back in time if necessary. The TRAIL POINTER as described in the unification section is saved in order to determine which variables in the stack must be unbound upon backtracking. The backtrack frame format is shown in Figure 35. The algorithm for the frame creation step is described in flowchart form in Figure 36.

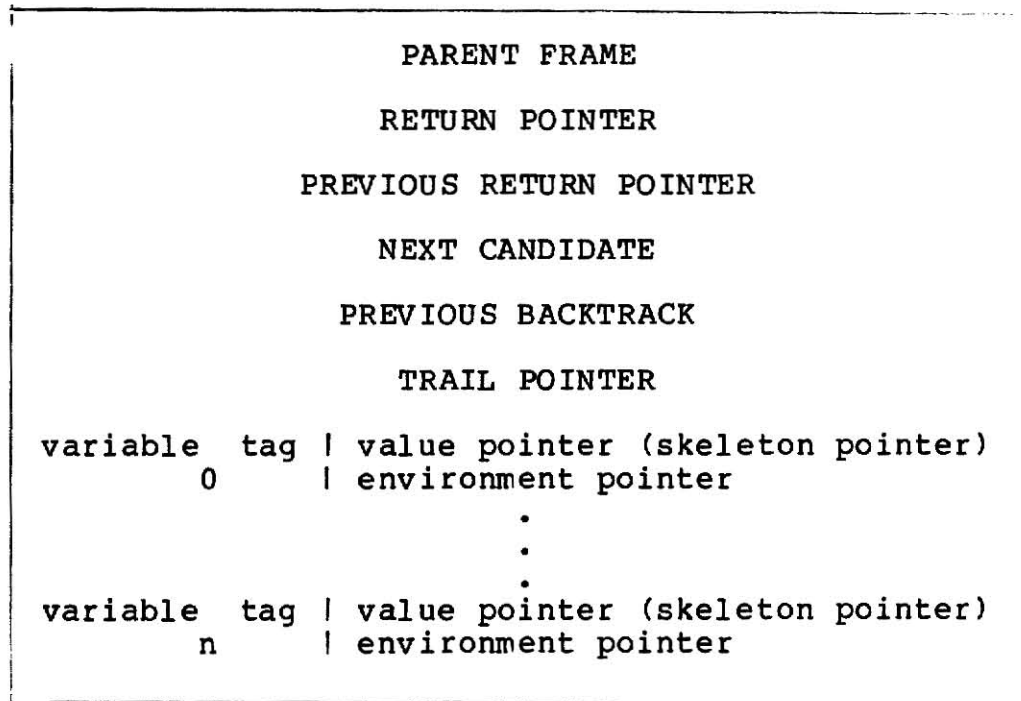


FIGURE 35

Backtrack Frame Format

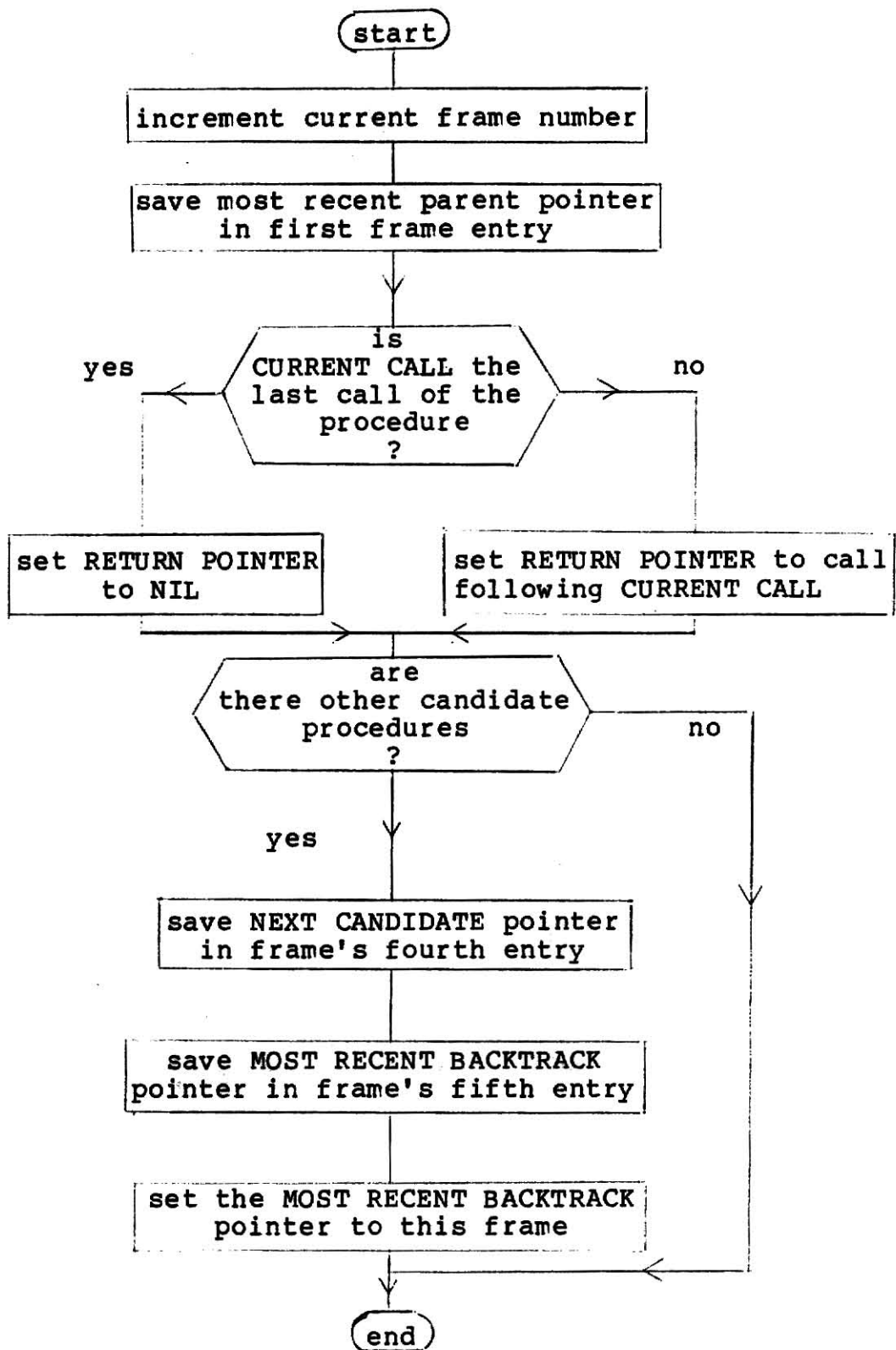


FIGURE 36

Frame Creation Flowchart

IX. BACKTRACKING

Backtracking occurs when a dead end is reached in the solution process due to no procedures being found to successfully unify with a current call. Backtracking may also occur after a goal has been completely solved and other solutions are desired to be found by exploring different solution paths.

When one of the above conditions exist, the interpreter examines the MOST RECENT BACKTRACK register to determine what frame was created at the last backtrack point. When this is determined, a pointer to the next candidate procedure to try in the unification process is found in the backtrack frame. The call to be solved at that point is found in the PREVIOUS RETURN POINTER of the frame, so CURRENT CALL is set equal to this.

All frames at and beyond the backtrack point are discarded because they now contain invalid data. Also, all variable tags pointed to by entries in the trail stack up to the saved TRAIL POINTER in the backtrack frame are reset to signify their being unbound because their values are invalid now. Finally, the global MOST RECENT BACKTRACK register is set to the saved PREVIOUS BACKTRACK pointer of the frame so that if further backtracking must take place, the interpreter can find the most recent backtrack point again. The flow of the algorithm is shown in Figure 37.

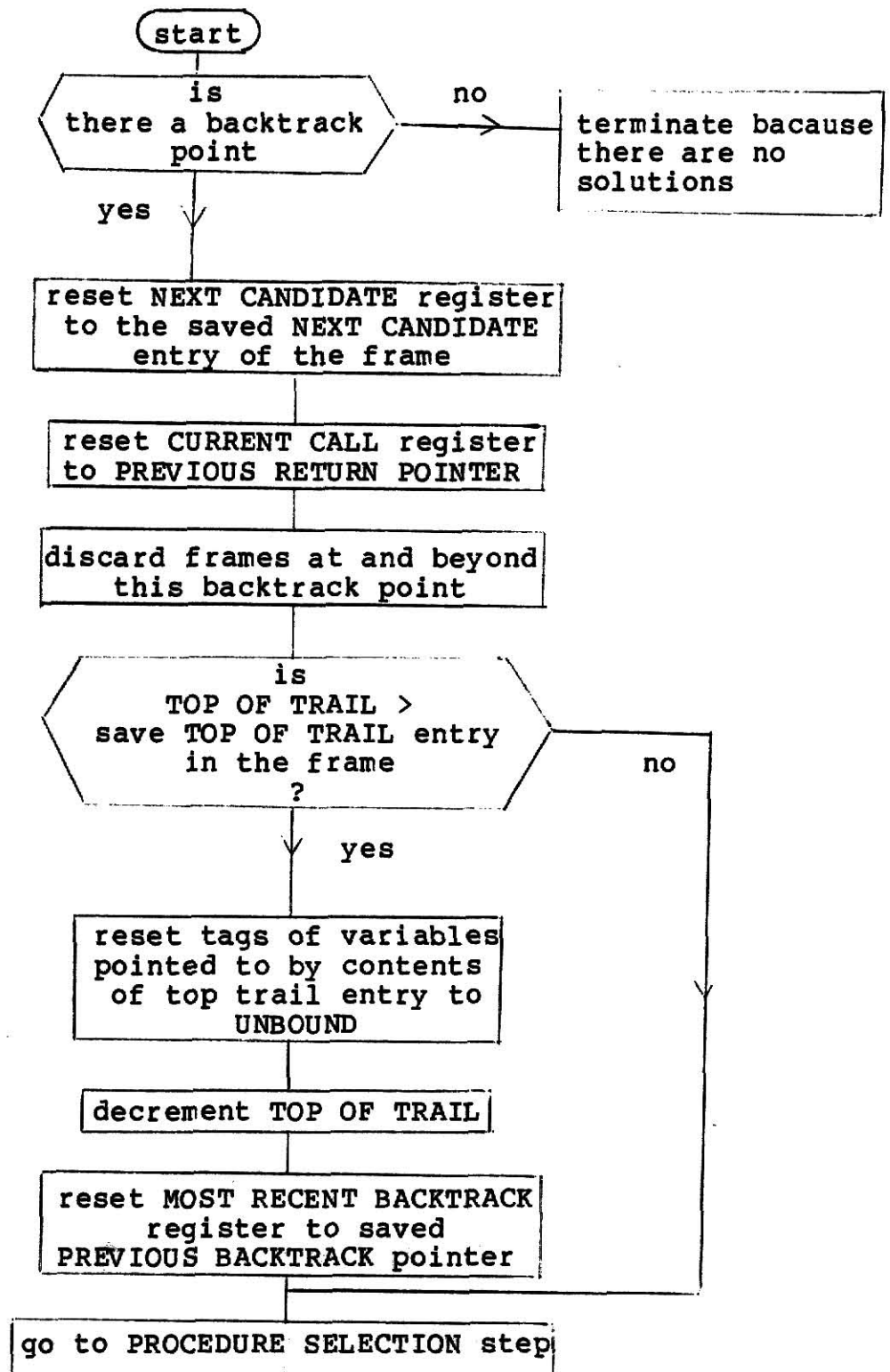


FIGURE 37
Backtracking Flowchart

X. SUMMARY OF INFERENCE ENGINE

The following pseudo-code outlines the control algorithm used by the interpreter to perform inferences. It is a modified form of an algorithm by Christopher Hogger.

Step 1 (Initializaiton)

```
frame := 1 ; The first frame is created
CURRENT PROCEDURE := -> input goal ; for the input goal, which
MOST RECENT PARENT := NIL ; is not a backtrack point.
MOST RECENT BACKTRACK := NIL
TOP OF FRAME STACK := -> frame stack
PARENT FRAME (1) := NIL
RETURN POINTER (1) := NIL
allot space for variables on frame stack
TOP OF TRAIL := -> trail stack
```

Step 2 (Call selection)

```
if CURRENT PROCEDURE -> is an assertion
then CURRENT CALL := RETURN POINTER (frame)
  while CURRENT CALL := NIL and MOST RECENT PARENT <> 1
  do
    CURRENT CALL := RETURN POINTER (MOST RECENT PARENT)
    MOST RECENT PARENT := PARENT FRAME (MOST RECENT PARENT)
  if CURRENT CALL = NIL
  then output goal solution
    go to backtracking step
  else CURRENT CALL := -> first call in CURRENT PROCEDURE->
else CURRENT CALL := -> first call in CURRENT PROCEDURE->
  MOST RECENT PARENT := frame

if no candidates exist for CURRENT CALL->
then go to backtracking step
else NEXT CANDIDATE := -> first candidate

; If the current procedure is an assertion, then
; the next call is found from the return pointers
; of the frames. Otherwise the next call is the
; the first call of the current procedure.
```

Step 3 (Procedure selection)

```
TRAIL POINTER (frame + 1) := TOP OF TRAIL          ; in case of
PREVIOUS RETURN POINTER (frame + 1) := CURRENT CALL ; a backtrack
                                                    ; frame
while no successful unification and candidates remain
  attempt unification between CURRENT CALL and candidate
  procedure.
                                                    ; variables are bound
                                                    ; during unification

if CURRENT PROCEDURE = NIL
  then go to Backtracking step      ; unsuccessful unification
  else go to Frame creation step    ; successful unification
```

Step 4 (Frame creation)

```
frame := frame + 1
PARENT FRAME (frame) := MOST RECENT PARENT

if CURRENT CALL-> is the last call in a procedure
  then RETURN POINTER (frame) := NIL
  else RETURN POINTER (frame) := -> call following CURRENT CALL->

if NEXT CANDIDATE <> NIL
  then
    NEXT CANDIDATE (frame) := NEXT CANDIDATE pointer
    PREVIOUS BACKTRACK (frame) := MOST RECENT BACKTRACK
    MOST RECENT BACKTRACK := frame

go to Call selection step
```

Step 5 (Backtracking)

```
if MOST RECENT BACKTRACK = NIL
  then terminate execution
  else
    NEXT CANDIDATE pointer := NEXT CANDIDATE (frame)
    CURRENT CALL :=
      PREVIOUS RETURN POINTER (MOST RECENT BACKTRACK)
    MOST RECENT PARENT := PARENT FRAME (MOST RECENT BACKTRACK)
    frame = MOST RECENT BACKTRACK - 1

    while (TOP OF TRAIL > TRAIL POINTER (MOST RECENT BACKTRACK))
      unbind variable pointed to by TOP OF TRAIL->
      decrement TOP OF TRAIL

    MOST RECENT BACKTRACK :=
      PREVIOUS BACKTRACK (MOST RECENT BACKTRACK)
    go to Step 3
```


To better illustrate the operation of the interpreter and the nature of the frame stack, a simple example will be shown. To help focus attention on the meaning of the global and frame pointers used, structure parameters are not used in this example.

The data base in this example consists of four procedures, along with the input goal of "likes(bob,Y)". The semantics of the inquiry can be interpreted as finding out who Bob likes.

```

                        Data Base
likes (bob, X) :- pretty(X), rich (X).
likes (bob, susan).
pretty (mary).
rich (ellen).
```

```

                        Inquiry
:- likes (bob, Y).
```

FIGURE 38

Database and Query for Interpreter Operation Example

In the initialization step, the first frame is built and CURRENT PROCEDURE is set to point to the facsimile representation of the input goal. The first frame has no parent. There cannot be any previous backtrack points, so the global pointer MOST RECENT PARENT and MOST RECENT BACKTRACK are set to nil. The trail stack is initially empty. The PARENT FRAME and RETURN pointer entries for the first frame are set to nil because there can be no parent for the first frame, and thus, no calls in a parent frame to return to. The input goal cannot be a backtrack point because there can only be one headless procedure in PROLOG.

The next step is the call selection step. Because the input goal is not an assertion, CURRENT CALL is set to the first call

of CURRENT PROCEDURE. Looking to the database for candidate procedures for "likes(bob,X)", it is seen there are two possibilities, "likes(bob,Y):- pretty(X), rich(X)" and "likes(bob,ellen)". The first of these becomes the candidate.

```

frame = 1
CURRENT PROCEDURE = -> :- likes (bob,Y)
TOP OF TRAIL → 

|                         |
|-------------------------|
| empty<br>trail<br>stack |
|-------------------------|


```

frame stack

```

frame 1 for
:- likes (bob, Y)

```

PARENT FRAME:	NIL	
RETURN POINTER:	NIL	
Y: UNBOUND	/	/

```

MOST RECENT PARENT = frame = 1
CURRENT CALL = -> likes(bob,Y)
NEXT CANDIDATE = -> likes(bob,X) :- pretty(X), rich(X).

```

FIGURE 39

Data Structures After Main Goal Frame is Built

The next step is the procedure selection, which attempts unification between the candidate procedures and the current call. The unification is successful in this case with "X" being bound to "Y". A frame is built. Because another untried candidate procedure exists, "likes (bob, ellen)", the second frame is a backtrack frame. The RETURN pointer is set to nil because the parent procedure (the input goal) has no other calls to solve.

Control passes to the call selection step. Here, the current call is set to the first call of the current procedure, "pretty(X)".

Unification is attempted with "mary" being matched with variable "X" of frame two, which is in turn bound to "Y" of frame one. A frame is constructed, but is not a backtrack frame because no other candidate procedures exist for the current call. A trail entry is made. The state of the frame stack at this point is shown in Figure 40.

```

MOST RECENT PARENT = 2
CURRENT CALL = -> pretty(X)
NEXT CANDIDATE = -> pretty(mary).

```

```

frame = 3

```

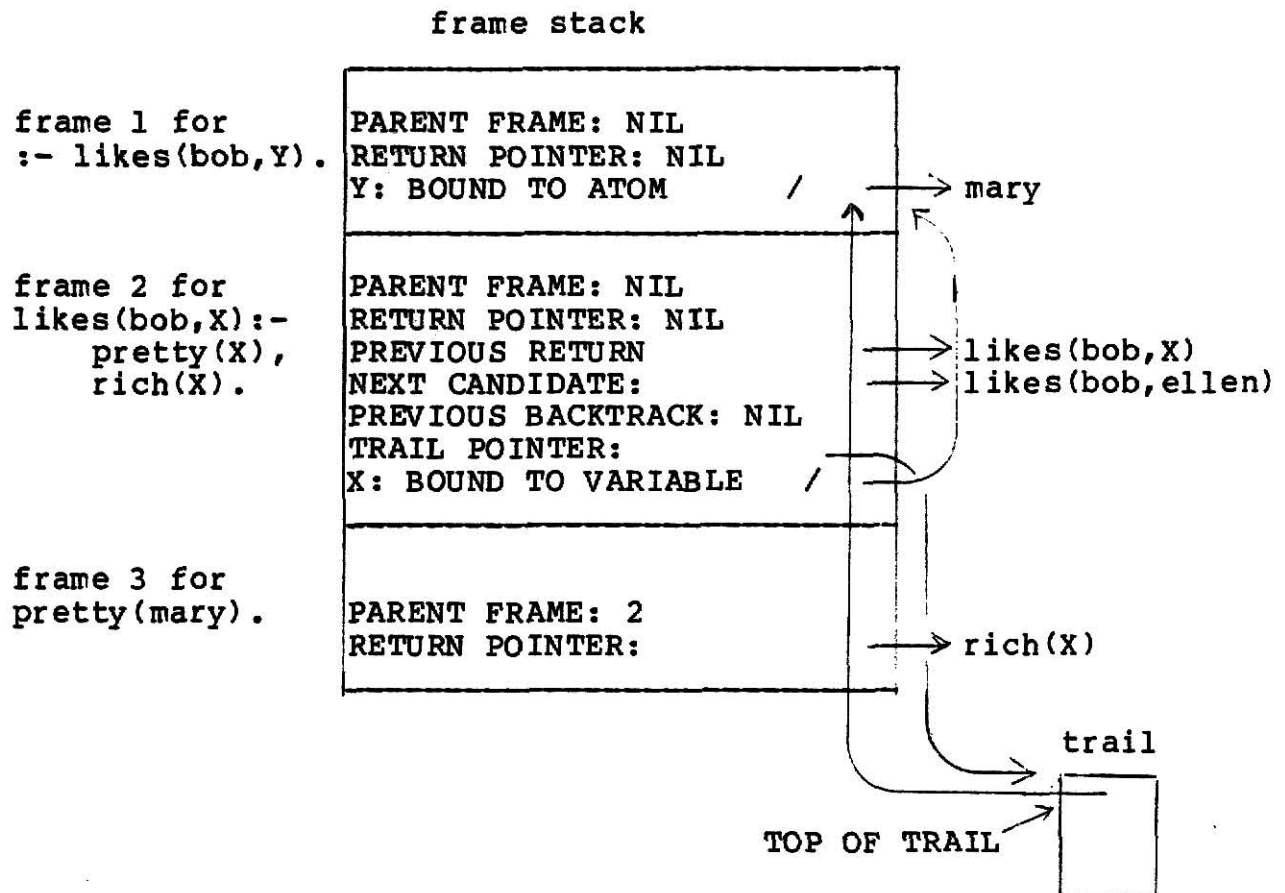


FIGURE 40

Frame Stack after Three Frames are Built

Control passes to the call selection step again. The current call is set to the return pointer of the current frame because the current procedure is an assertion. Searching for candidates for "rich(X)" results in finding "rich(ellen)". However, when unification is attempted during the procedure selection step, the parameter of the current procedure will not match with the variable parameter of the current procedure because it is bound to "mary". Thus control passes to the backtracking step.

A backtrack point exists, occurring at frame two according to MOST RECENT BACKTRACK. The next candidate is set to NEXT CANDIDATE of frame two. The current call is set to PREVIOUS RETURN pointer of that frame also. The trail variables are reset, with the trail pointer restored to its saved value. Now MOST RECENT BACKTRACK is set to nil because no more backtrack points exist. Frames two and three are discarded, leaving the state shown in Figure 41.

```

frame = 1
MOST RECENT BACKTRACK = NIL
CURRENT CALL = -> likes(bob,X)
MOST RECENT PARENT = 1
NEXT CANDIDATE = likes(bob,susan)

```

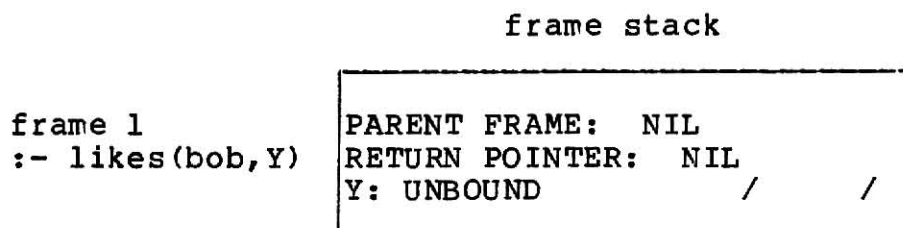


FIGURE 41

Frame Stack After Backtracking

Unification is attempted and successful for the next candidate and current call. "susan" gets bound to variable "Y".

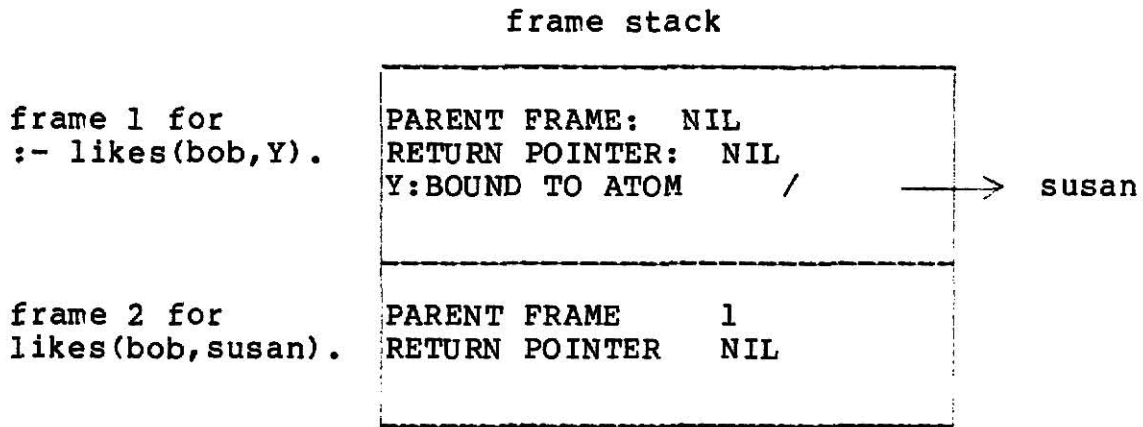


FIGURE 42

Frame Stack after Goal is Solved

When control passes back to call selection, no more calls are found and no more backtrack points exist, so the solution is output and execution terminates.

XI. CONCLUSIONS

The principle effort of this report was the description of an algorithm and the associated data structures for a PROLOG interpreter. A scheme for encoding the PROLOG database in a form that is compact and conducive to quick unification was described. It is this part of the interpreter that is most sparsely documented in the literature.

The algorithm and data structures presented provide sufficient groundwork for actually implementing a PROLOG interpreter. Moreover, the interpreter described gives some insight as to a computer architecture on which to run PROLOG more efficiently. Considerable use is made of tagged data fields and stacks, and thus are appropriate foci for future research of a suitable architecture.

References

1. Sowa, John. Conceptual Structures. Reading, Massachusetts: Addison-Wesley, 1983.
2. Mota-oka, T. Fifth Generation Computer Systems. Amsterdam: North-Holland Publishing Company, 1982.
3. Rich, Elaine. Artificial Intelligence. New York: McGraw-Hill, 1983.
4. Hogger, Christopher. Introduction to Logic Programming. London: Academic Press, 1984.
5. Clocksin, W. and Mellish, C. Programming in PROLOG. Berlin: Springer-Verlag, 1981.
6. Chang, C. and Lee, R. Symbolic Logic and Mechanical Theorem Proving. New York: Academic Press, 1973.
7. Cambell, J. Implementations of PROLOG. Chichester, England: Ellis & Horwood Limited, 1984.
8. Kruse, Robert. Data Structures and Program Design. Englewood Cliffs, New Jersey: Prentice Hall, 1984.

A CONTROL STRATEGY FOR A PROLOG INTERPRETER

by

DAVID J. RODENBAUGH

B.S., Kansas State University, 1984

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

1985

ABSTRACT

PROLOG has been the focus of much interest in the area of artificial intelligence. It has been selected by the Institute for New Generation Computing Technology as the kernel language for their fifth generation computer system. This has sparked interest in the language and computer architectures tailored to it. This report describes the inference mechanism of a PROLOG interpreter in terms of algorithms and data structures. The approach is based on a scheme by Hogger. The majority of the effort is concentrated on describing the data structures for encoding the PROLOG database in a form that is compact and conducive to efficient solution search. The interpreter described suggests that tagged architectures and stack architectures merit more research as possible architectures on which to run PROLOG more efficiently.