1986

Data Structure Implementation and Investigation for a Prolog Language Interpreter

Hector Iurcovich
University of Central Florida

DATA STRUCTURE IMPLEMENTATION AND INVESTIGATION FOR A PROLOG LANGUAGE INTERPRETER

BY

HECTOR IURCOVICH
B.S.E., University of Central Florida, 1985

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Graduate Studies Program of the College of Engineering
University of Central Florida
Orlando, Florida

Fall Term
1986

ABSTRACT

A developmental Prolog language interpreter is constructed in the C programming language. The data type format is a subset of that of the Project Aquarius Prolog Engine proposed and under development at the University of California, Berkeley. The design of the Prolog interpreter is part of an investigation at UCF in design of expert system architectures for real time applications.

The interpreter facilities of data structure definition and manipulation are developed and applied. The list data structure is used as an example data structure.

The scheme for representing structured data is based on the Lisp cons cell. Run-time generated lists are represented by car and cdr pairs and are maintained separately from data structures which are part of the Prolog program.

A data memory is used to augment the environment memory of the Prolog interpreter. The advantages and disadvantages of the proposed structure and implementation scheme are discussed.

ACKNOWLEDGMENTS

The author wishes to express his sincere appreciation to his major professor, Dr. Brian Petrasko, for his guidance and encouragement during this study. The author would also like to extend his gratitude to Dr. Harley Myler and Dr. Robert L. Walker for serving on the committee and reviewing this report.

Thanks are especially due to Ms. Candie Steward, editorial assistant University Library, for her help in the proper preparation of this thesis.

My special thanks to my family for their support which enabled the author to complete his study program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Prolog is an accepted and developing language, being used in the design of many expert systems. The Japanese are using it for fifth generation systems design and it is the base language of knowledge information processors being designed at the University of California, Berkeley. Prolog is now available on the personal computer and is being discussed in the "small system" environment. This will increase its popularity and acceptance.

A research effort directed at the design of special purpose prolog-based engines for application in real time systems is underway at the University of Central Florida. At present an object code interpreter P5, is specified for the Prolog development project.

The objective of this study is to add a data structure facility to the object code interpreter.

The following chapters will overview the Prolog language and discuss the architecture of the interpreter. Chapter II is a description of the interpreter. Chapter III presents the design of the data structure facility under the current architecture.

Chapter IV, Summary and Conclusions, contains suggestions for the modification and extension of the architecture.

Expert Systems

Expert systems are used to solve problems of realistic complexity with relative high performance. They should also be able to explain the solution. Most expert systems consist of a database to hold knowledge, and an inference engine used to process the knowledge base and generate solutions.

Computer based problem solving can be based on "State Space Search." This scheme entails starting with an initial state, testing for a final state or goal state, and if failure, then generating and testing new states using available rules and operators. The search method used could range from a depth-first search to a breadth-first search or some guided depth-first search using an evaluation function.

Current research and development efforts in expert systems focus on Pattern Directed Inference systems (PDI). These are systems where knowledge is represented by "formalisms" called rules or structures or clauses which are searched by the inference engine for patterns associated with the current state and the goal state.

Current issues in PDI systems include knowledge representation and performance. Knowledge is represented as rules sets (production systems), generalized graphs (association nets or object oriented systems) and predicate logic (logic programming systems).

An important element in performance is the use of strategies or heuristics to reduce the space search domain. Logic programming systems, such as Prolog, have a fixed strategy which is simply the ordering of rules and facts.

Production systems provide for alternative strategies. These systems consist of a rule set, working memory and an interpreter. Rules are condition-action pairs. The conditions are object-attribute-value (OAV) triples; these are the "patterns" referred to in the definition of Pattern Directed Inference systems. The actions will alter working memory, perform computations, and do input/output. Working memory contains data (OAVs) and the data structure which is modified. It also contains the goal. The interpreter operates in a recognize-act cycle. It matches conditions, selects the rules, and terminates on no matches. Some of the advantages of these systems over Prolog are that the interpreter views all matches and selects the next rule using conflict resolution strategies, whereas the Prolog strategy is to select the first valid match in a search of the database.

The conflict resolution concept embodies the use of meta-rules.{10} The addition of strategies in logic programming systems is a current research effort.{9}

Overview of Prolog Terminology and Processing

The Prolog language is an interactive language founded on symbolic logic and based on a formal mathematical system from a subset of classical logic. It performs its computations by pattern matching over its data elements, a process called unification.

The unification of items is successful if the items (and the data elements associated with the items) are the same or are compatible. Thus a set of data elements unify if, given the relationship between these data elements, there are no conflicts with respect to their value. Successful unification can result in the binding of data elements.

Prolog is a language that is simple and easy to learn yet it is powerful for many applications {8}. Most current Prolog implementations use the following terminology and definitions. A constant is an alphanumeric symbol that begins with a lowercase letter. A variable is represented by an alphanumeric symbol that begins with an uppercase letter.

A clause is a primitive Prolog expression. A clause has a head and may have a body. The clause head has a name, predicate or functor, and arguments. Clauses with the same name constitute a Procedure.

A clause without a body is referred to as a fact. The functor is a relationship and the arguments are ordered elements of the relationship. The following is an example of a fact. The first argument is the father, the second argument is the son.

```
father(robert,gary).
```

A clause with a body is referred to as a rule. The term rule is used to represent a clause which has both a head and a body.

For example, the following is a rule that could be part of the database.

```
grandfather(X,Y) :- father(X,Z), father(Z,Y).
```

The head of this rule (clause head) contains two arguments, X and Y, which are two of three local variables used in the rule expression. When the rule is activated as a result of unification of the rule (clause) head with a calling clause head, the local variables in the rule head

are bound to the arguments (constants or variables) of the calling clause head.

The body of the rule consists of calls to the database for rules and facts. The subsequent unifications bind the unbound local variables. On termination of the rule, marked by the period (.), the bound local variables are bound to the calling variables and thus the unification of the calling rule head is completed.

The term goal is associated with the calling clause head and the body consists of subgoals. For the goal to be met, all the subgoals must be met.

The body contains two subgoals and an additional local variable; Z. This local variable links the two clauses of the body. The second father clause in the body will use the value bound to Z by the first father clause.

Arguments in Prolog are in general called terms; these can be constants, variables, or as discussed in Chapter III, data structures, which are compound terms.

In P5, there are privileged constants. The constant NIL is used to delimit the end of a clause or data structure. A variable can become bound to a constant, another variable or a data structure. Variables are local to rules and are essentially indices. A data structure is an object which consists of a functor followed by a sequence of terms as arguments. This is discussed in detail in Chapter III.

Unification Process and Backtracking

Unification is the pattern matching method used by Prolog to bind values (i.e., constants).

Many different unification cases arise due to the possible types of data words. For example, two constants unify only if they are the same constant. If a variable has not yet been bound then it will unify with a constant or another variable. Two data structures will unify if all of the respective subelements of the structures unify.

All information regarding bindings must be kept in memory. This is necessary for returning a value for the calling variables of a calling rule, and for backtracking.

Backtracking occurs when, during unification, a fact or a rule cannot be found that unifies with a subgoal. Therefore the interpreter must return to its last successful unification, called a choice point, and undo the bindings that caused it to fail; and retry the unification of the goal (subgoal of the present rule) with the next rule or fact in the database with the same functor.

II. THE PROLOG INTERPRETER

The main objective of the interpreter is as a development tool. The long-term objectives are to examine the partitioning of an inference engine into functional components that can be optimized for specific applications. For example the control component can include additional search strategies instead of the standard top-down left-right Prolog scan of the database. Other components that can be added and/or optimized include the addition of pre-fetch of rules and capabilities for tracing the solution process.

The P5 Prolog interpreter is an object code interpreter in which memory is structured as strings. A list processing system, XLISP, is being investigated as a host system. The host system will provide facilities for Prolog source code translation, I/O, and memory management.

The Prolog interpreter expects its data organized in tables and string memory. This data is generated by a Prolog source compiler. In this version the memory is declared and initialized in a header file. The following sections will describe the data types used by the interpreter and its memory organization.

The following is a block diagram of a Prolog system.

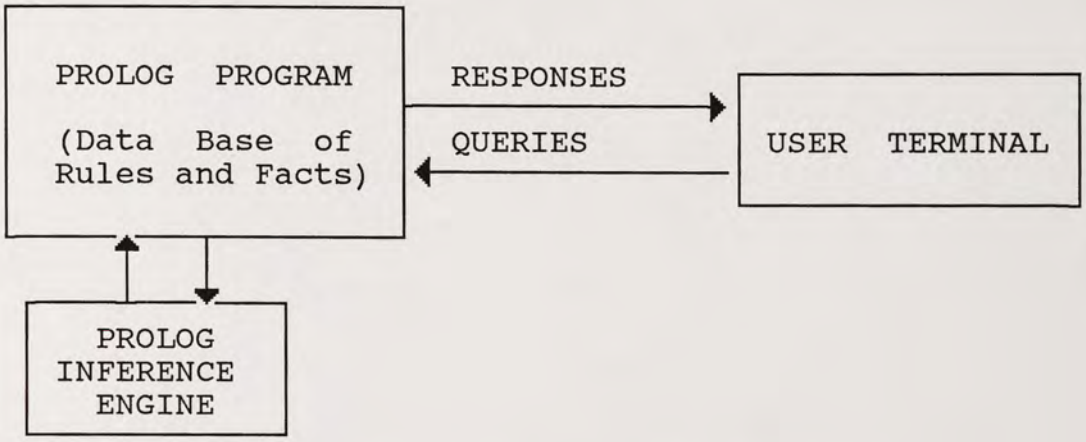


Figure 1. Block Diagram of a Prolog System.

The P5 interpreter uses the following data word format: Words are 16 bits in length where the four most significant bits are used for the type. The twelve least significant bits are called the value. The value field may have subfields as in the case of structures. In this case there is an eight bit value field (the functor) and a four bit arity field. The following figure shows data word format of P5.

Type[15:12]	Value [11:0]

Figure 2. Word Format in P5.

The following table describes the data type assignments for the P5 interpreter.

TABLE 1. DATA TYPE CODES

TYPE [15,12]	CODING	HEX VALUE
Structure	11XX	
Clause (SC)	1100	C
Data structure (SD)	1101	D
Built-in (SB)	1110	E
Procedure (SP)	1111	F
Reference	01XX	
Variable (RV)	0100	4
Pointer (RP)	0101	5
Reference Pointer_Fact	0110	6
Reference Pointer_Rule	0111	7
Constant	00XX	
Constant (C)	0000	0
Constant_NIL (NIL)	0001	1
Constant_EOS (EOS)	0010	2
LINK (NOT USED)	10XX	

The following illustrates the Value Field for Data Types:

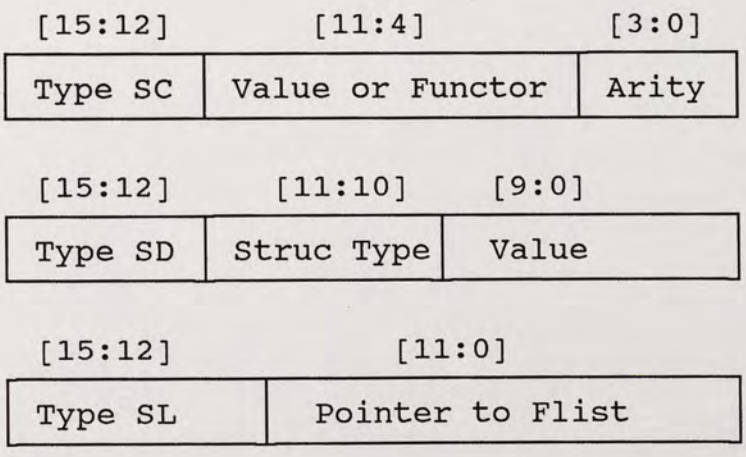


Figure 3. Value Field Format for Data Types.

P5 accesses a number of memories. Table 2 presents an overview of the P5 memory organization.

TABLE 2. P5 MEMORY ORGANIZATION

MEMORY AREA	FUNCTION
M[]	STRING MEMORY WITH FACTS AND RULES DATA ELEMENTS ARE 16 BITS IN LENGTH ACCESSED BY GSP AND RSP POINTERS
STB[]	STRUCTURE TABLE CONTAINS POINTERS TO M[] AND FLIST[].
FLIST[]	USED FOR PROCEDURES. ELEMENTS ARE POINTERS TO M[] MEMORY.
A[]	A MEMORY USED FOR CONTROL AND BINDING INFORMATION.

The M memory is implemented as a string and it contains rules and facts and is terminated by the End_of_String or EOS data word.

A rule consists of rule head arguments (single data words) followed by subgoal(s). Each subgoal begins with a Structure_Clause data word, which includes the functor and its arity (number of arguments), and is followed by the arguments of the subgoal. The rule is terminated by the privileged constant NIL.

A Query or initial goal(s) is translated to the subgoal format and is appended to the end of the M memory string.

The Structure Table memory, STB, is used to locate rules and facts in M memory. It contains data types Structure_Rule, Structure_Fact, and Structure_List. In P5, STB is a table structure which is accessed by the pointer (functor) in the subgoal SC data word. For cases where the facts or rules have the same functor or search key, it is necessary to not only search STB but also FLIST.

The FLIST memory is used in the case of procedures. A procedure contains clauses with the same clause head functor. The structure table, STB, is linked to the string of pointers in FLIST by the data type Structure_List.

The A memory or rule activation memory contains the state and binding information. It is a linked record structure. Activation records are terminated by a NIL data word.

The A memory records contain a header field where information for backtracking is kept. It also contains the argument binding records. These records contain four fields C,V,G, and F.

The C field normally contains the bound value or constant, the V field contains the bound variable, the G or Goal field contains the pointer to the head of the calling

clause or subgoal, and the F field contains zero or an index of the FLIST string. Thus the F Field marks (by dereferencing) the match clause in the program string. If F is zero, or the index points to NIL in FLIST, there are no further matches and the goal fails.

Comparison of P5 to Project Aquarius

The UCF Prolog interpreter can be compared to other developmental Prolog systems. The following statements address the differences and similarities of the UCF Prolog interpreter P5, and the Prolog system of the Aquarius project at University of California at Berkeley.

In the Project Aquarius {3}, the Prolog source code is translated to an instruction set proposed by Warren{1}, which includes instructions such as get_c, unify_c and call_p (procedure). This is assembled and is resident in the host computer, an NCR/32, in an area called the code space. It is accessed by the Prolog Engine {6}. The Aquarius system has data types Reference, Constant, Structure and List. The data type Reference is associated with a variable and points to data words which are bound to the variable. The data type Constant holds a value and has a field which specifies the

type of value. The Structure and List data types are used to point to values in the heap in the data space. All data words are cdr coded. In the cdr coded scheme a bit of each data word indicates whether the current element is a car or a cdr. Thus lists can be stored sequentially in memory by just using car elements, a cdr element is used when a discontinuity is present and the reminder of the list needs to be indicated. This is a typing of each structure element so that it can be processed as a list.

The data space for Aquarius is in the host and consists of various structures, namely, Environment Frames, Choice Point Frames, Trail Stack and Heap.

The A memory in P5 is used for all of the above except the Heap. This is the data structure memory (DM) which is added to P5 in Chapter III.

Interpreter Operation

The interpreter is presented a query which is a goal(s). The goal is appended to the program string in M memory and an activation record is created in the A memory if the goal is matched with a rule in the program string.

The found rule is processed. The subgoals of the rule string constitute new goals, each of which must be unified. Each subgoal or new goal which is a rule adds goals to the search process.

This "stacking of goals" is implemented by the structure of the A memory. When a match is to a rule the A memory is "pushed." A new activation record is created and once the rule has been unified, the A memory is "popped."

The interpreter continues processing until the A memory is "popped" to the query activation record. At this point the query has been unified with the program and the bindings, which have been made during the process, are in the query activation record. This is presented to the I/O facilities as a response to the query.

A virtual goal stack is implemented by storing the goal string pointer as a reentry point to the program string, in the A memory.

Interpreter Functions

P5 is written in the C Language. The interpreter is organized as functions with specific tasks. The major global variables are the goal string pointer, gsp, the activation memory pointer, ap, a flag of the type of clause under

match, rule flag (rf), and the pointer to the clause (rule or fact) under match, Rule_String_Pointer (rsp). Additional global variables are also used.

The main() module accesses the goal element pointed to by the gsp (this is initially set at the time the query is appended to the program string). The goal element is examined for type and value. Depending on the goal element type the following actions are taken :

If the goal element is a structure clause, SC, then fetch() is called to find a matching rule or fact in the database. Fetch sets the rule string pointer, rsp, and returns a value to indicate a matching rule element was found. Fetch also determines whether the match was a rule or a fact (rf). In the case of a rule, new subgoals need to be satisfied; therefore push() is called to create a new activation record in A[] memory.

The arguments of the goal string and the head arguments of the rule string must be unified. The module unify() is called for this purpose.

Other possible goal elements examined by main() are NIL and EOS. The element NIL leads to a pop() since it indicates the end of the current goal string. The EOS element is found when the virtual goal stack is empty, therefore no more processing is required.

The interpreter displays state information at this point for debugging and tracing purposes.

The fetch() module will try to find a matching functor for the current goal. The current implementation uses the goal element value and arity as a pointer to the Structure Table, STB. This table contains elements which are reference pointers (RP_R or RP_F) to rules or facts in M memory, or the Structure_List data element, SL. The latter indicates that there is a number of matches. The value field of the SL element is a pointer to FLIST memory and leads to a list of pointers to the requested functors in the M memory. The SL value is the fact rule marker (frm) and is used to remember the position in flist of the current match. The frm is used in the construction of a binding record in the A memory.

Fetch will also set a global variable rf to indicate that it has found a rule or a fact. Upon return to main(), the global variables arity (arity), rule string pointer (rsp), and the flist marker (frm) are set.

The unify() module tries to unify the arguments pointed to by the gsp and rsp. The number of arguments to be unified is given by the arity of the functors. The variable bindcnt is used to remember the number of bindings resulting from the current unification. This is necessary in case of

non-unify. In unify(), the elements are examined for type and value. If the goal element is a reference variable, and it is bound, then the goal element takes this bound value. A five bit key (fkey) is used to select the cases that can be considered for unification. The key is formed by combining the rule flag (rf) and the first two type bits of the goal element and the rule element.

The following table shows the possible unification cases that are handled by the interpreter.

TABLE 3. UNIFICATION CASES

UNIFY	FKEY	TYPE
FCC	00000	Fact Constant Constant
RCC	10000	Rule Constant Constant
FCV	XXXXX	{No variables in Facts}
RCV	10001	Rule Constant Variable
RVV	10101	Rule Variable Variable
FDD	01111	Fact Structure Structure
RDD	11111	Rule Structure Structure
FDV	01101	Fact Structure Variable
RDV	11101	Rule Structure Variable
RVD	10111	Rule Variable Structure
FVD	00111	Fact Variable Structure

P5 is a flexible system with a lot of room for expansion and improvement. The following chapter describes the data structure addition to P5.

III. THE DATA STRUCTURE FACILITY

The data types in the initial P5 interpreter are variables or constants which are represented by a single data word. There is no global storage, which is consistent with the concept of a functional language. Each rule, upon activation is passed calling arguments which consist of single words. Upon termination the rule returns arguments which are also single words. Modern versions of Prolog {8}, and the Project Aquarius {3}, provide for data types of compound objects and structures, respectively.

Implementation of a Data Structure for P5

A data structure facility for P5 is proposed. Data structures which are included in the source code are translated to a data structure format, and placed in a new memory, Data Memory or DM. The pointer to the structure position in DM is placed in a table, Data Structure Table, DSTB. The table address, or the Structure Key, is made the Value field of a new data type Structure_Data.

It is anticipated that a number of special data structures will be useful in a Prolog environment. One such

structure is the LIST structure. Thus the Structure Data type is designed with a type field.

Implementation of the List Data Structure in P5

The common syntax and semantics associated with lists is to be added to P5 source code notation for arguments with respect to the following source code examples. The object code representation of a list is discussed below.

An argument which is a list of three elements can be represented as [A.B.C]. using the infix notation, the functor is the ".". The empty list is denoted by the element [NIL].

A data structure can be implemented as a block of contiguous words (a string) in a data memory space. This would be the same as the format for a clause structure which is a fact (i.e., father(a,b)). Note that the functor and arity would be a Structure_Data word which is stored in a table.

Another more common way to think of a list is as a structure of arity two, whose functor is the ".". The first argument is a value, the head of the list, and the second argument is a pointer to the remainder or tail of the list. The first or second argument may also contain the constant NIL to indicate an empty list or a list of one element.

In list processing systems, lists are usually processed from head to tail using a recursive procedure that accesses the elements of the list until the element NIL is reached. In P5 List type data structures are found in two forms, a string format, and a list format.

In the string-based format a list structure is represented as a block of contiguous words in a data memory. A Structure type word points to a string of data words which can be constants, variables or other lists. Thus under this scheme lists are represented as a string of data words.

In the list-based scheme the primary structured data element is a list cell similar to the cons cell {7}. The cons cell contains two elements, the car and cdr. The car is the current head of the list and the cdr is the tail of the list. Thus a list structure is represented as cons cells which are linked by their cdr elements. A list name (the Key) is assigned for each List data structure and stored in a structure data table. The list name or the Key is used to reference the first cons cell in data memory. This scheme requires more storage necessary for representing list structures than the string-based scheme. These schemes are displayed in figures 4 and 5. The indirection through the table DSTB is shown as a broken pointer. For small systems the functor can be replaced with a pointer since a name or key is not needed for a list data structure.

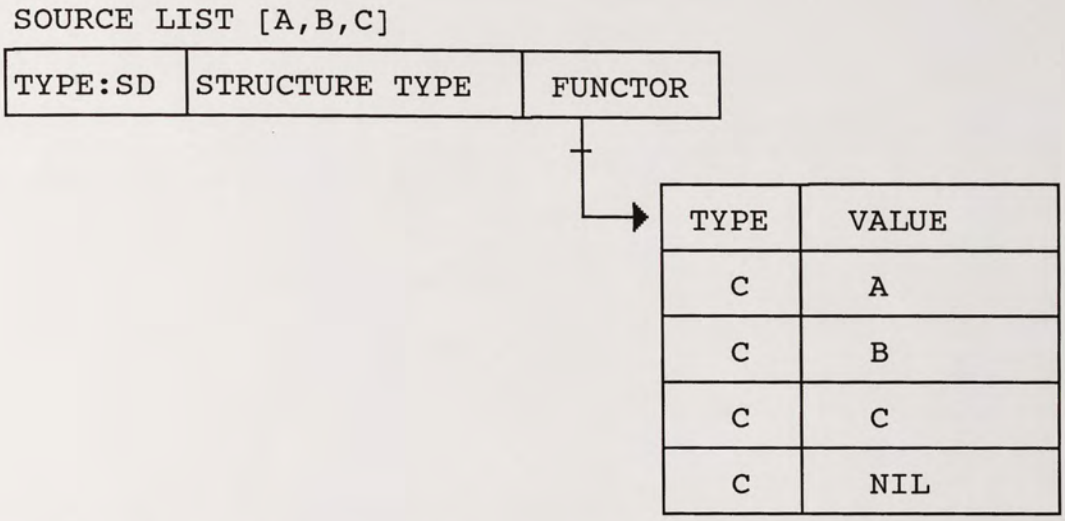


Figure 4. String-Based Representation of a List in P5.

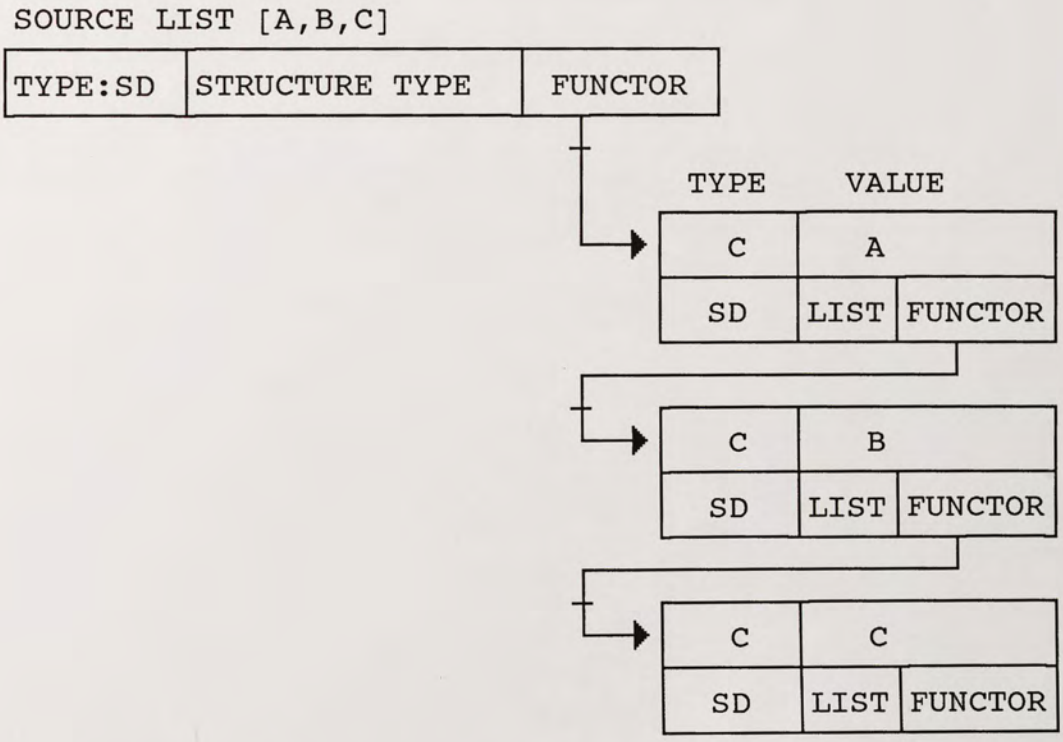


Figure 5. List-Based Representation.

In the current architecture source lists are stored in data memory DM as strings. The lists can have any number of words and must be terminated by the constant NIL. Any new (run-time) lists are two-word (tuples or word pairs) which are based on the Lisp cons cell.

In Common Lisp {7}, a-lists or association lists have a number of attributes which would be useful in Prolog. An a-list is a list of conses, where each cons is an association. The car of the cons is called the key, and the cdr is called the datum. Association lists provide for nested structures.

In the run-time lists the car can be a constant, variable or structure data word and the cdr can be a constant, variable or a structure data word.

An example of list creation will be presented in which the final form of the list consists exclusively of cons cells. This type of list has the worst case cost in terms of dereferencing. The source code list structure has the best case dereferencing cost.

Data Structure Format in P5

Source code structures are translated to a Structure_Data data word followed by argument data words.

The format for a Structure_Data data word is:

[15:12]	[11:8]	[7:0]
Type: SD	Structure Type	Structure Key

For the List type, no Structure Key is given in the source code, but is assigned by an I/O process. The object form of a List structure is a string in data memory, DM, and a Reference Pointer in a data structure table, DSTR. The structure Key which was assigned is used in the Structure_Data data word and the list is represented by this data word.

The following shows the contents of Data Memory for a Structure List [a,b,c].

```
dm  = {
  0X000A  /* C, A First element  is constant   A  */
  0X000B  /* C, B Second element is constant   B  */
  0X000C  /* C, C Third element  is constant   C  */
  0X1000  /* NIL  End of List                     */
 }
```

The following table describes the new data memory facilities added to P5.

TABLE 4. DATA STRUCTURE MEMORY

MEMORY AREA	FUNCTION
DSTB[]	DATA STRUCTURE LOOKUP TABLE CONTAINS POINTERS TO A DATA STRUCTURE IN DM[] MEMORY
DM[]	DATA STRUCTURE MEMORY: LISTS AND STRUCTURES ARE SUPPORTED IN THIS AREA
BUFFER[]	MEMORY USED FOR RUN-TIME GENERATION OF DYNAMIC DATA STRUCTURES

The Data Structure Table memory, DSTB[], contains elements of type Reference_Pointer, RP. This table is used to store the pointers to Data Structure Memory, DM. The Buffer Memory is used to temporarily store the car and cdr pairs for run-time generated data structures. Since the Data Structure Table and Data Memory may need to be updated at run-time, two additional variables, dmend and dstbend, are defined to indicate the current end of these memories.

Data Structure Facility Operation

During unification, the unify() module examines the types of the arguments of the goal string and rule string.

If the element types are both data structures, the module dp() is called to access the data structures using the data structure dstb[] (lookup) table. The results are pointers to the first arguments of each structure in dm[] memory. The dp() module informs the calling module if the two data structures unify.

The unification is performed by the Data Structure Unify module, du(). Unification of the list elements proceeds until the element NIL is reached.

The operation of the interpreter is presented via an example. This example demonstrates the creation of lists at run-time.

The Makelist Program Example

```
Procedure:    makelist(X,Y).
     (M1):    makelist(1,[1]).
     (M2):    makelist(N,[N|REST]) :- decrement(N,N1),
                                      makelist(N1,REST).

Procedure:    decrement(X,Y).
     (D1):    decrement(3,2).
     (D2):    decrement(2,1).
```

The program steps in the makelist example are included in Table 5.

A Query, makelist(3,L)? starts the session (Step 1). The source code is translated to object code and appended to the program string by the I/O processor. The procedure makelist is referenced and the first clause of makelist, M1, is unified with the query (Step 2). This will fail because the constant 3 is not equal to the constant 1. The interpreter will look for another clause in the makelist procedure. It will find M2 which is a rule. The rule head arguments will be unified. Since the match is a rule, new subgoals will have to be satisfied and a new activation record is initialized by the push() module (Step 3). The unification succeeds and the following bindings are added to the new activation record in A memory: (3/N) or constant 3 to reference variable N and (L/SD2) or calling variable L to a Structure_Data, SD2, which represents the elements [N|REST]. The first subgoal of M2 is examined and D1 is accessed (Step 4). The unify module will attempt to unify decrement(N,N1) with decrement(3,2). Since the goal element is a reference variable and is bound to 3, the first two arguments unify. The second arguments also unify and a binding record is added to the current activation record (2/N1) constant 2 to reference variable N1.

The second subgoal of clause M2 is examined and clause M1 is accessed. The unification of makelist(N1, REST) with makelist(1,[1]) will fail since N1 was bound to 2 (Step 5). The interpreter will select clause M2 as the match. M2 is a rule and Push() is called to create a new activation record. The rule head arguments are unified as makelist(N1,REST) with makelist(N,[N|REST]) (Step 6). This passes and the following bindings are added to the current activation record, (2/N) and (REST/[N|REST]) where [N|REST] is SD2.

Again the subgoal decrement is examined and D1 is accessed. This fails since N is bound to 2 (Step 7). The next clause in procedure decrement, D2, leads to unification of decrement(N,N1) with decrement(2,1). This passes since N was bound to 2 (Step 8). A new binding, (1/N1) N1 bound to 1 is added to the current activation record.

The next subgoal leads to unification of makelist(N1,REST) with makelist(1,[1]). Since N1 is bound to one, this clause in procedure makelist passes (Step 9). The resulting binding (SD1/REST) is added to the current activation record. SD1 contains [1.NIL].

With successful unification of makelist, the end of clause M2 is reached (Step 10), the element NIL is found and the pop() module will identify the calling variable, REST, in the current activation record. Rest is bound to SD2.

The pop() module will call buildata() and analyze the contents of SD2 which is [N|REST]. The bindings for these local variables, (2/N) and (SD2/REST) are obtained and a new data structure, SD3, is created. SD3 is a cons cell with car = 2 and cdr = SD1. SD3 is created with N=2 and REST = SD1 = 1. This new data structure is stored in dm memory. This results in a new binding record with calling variable REST bound to a created data structure SD3. Note that SD3 = 2,SD1 and SD1 = 1,NIL. The current activation record is "popped" and the returned binding is appended to the previous activation record.

At this point the interpreter recognizes a second pop(), (Step 11) due to the end of the first clause M2 called. The first activation record is searched for calling variables. It is found that L is bound to SD2 where L is the calling variable. The module buildata() examines the contents of SD2 and finds [N,REST]. The bindings for these local variables are: N bound to 3 and REST bound to SD3.

A new data structure, SD4, is created. SD4 is a cons cell with car = 3 and cdr = SD3. Note that SD3 = 2,SD1 and SD1 = 1,NIL. A binding record is constructed with L bound to SD4. This is returned to the Query activation record.

The end of the query string makelist(3,L)? is the end of the program string. The I/O processor is called and the response L = [3,2,1] is output.

It should be noted that in Turbo Prolog, an infix notation for decrement is used. Built-Ins are special functors that would normally be used to implement the decrement procedure. The module main() would recognize the Built-In data type (SB) and call a host system routine.

TABLE 5. PROGRAM STEPS AND RESULTING BINDINGS

STEP	ACTION	GOAL	RULE/FACT	BINDINGS
1.	Query	ml(3,L)?		
2.	Unify	ml(3,L)	ml(1,[1])	FAIL
	Push			
3.	Unify	ml(3,L)	ml(N,[N|REST]	3/N L/SD_2
4.	Unify	dec(N,N1)	dec(3,2)	2/N1
5.	Unify	ml(N1,REST)	ml(1,[1])	FAIL
	Push			
6.	Unify	ml(N1,REST)	ml(N,[N|REST])	2/N REST/SD_2
7.	Unify	dec(N,N1)	dec(3,2)	FAIL
8.	Unify	dec(N,N1)	dec(2,1)	1/N1
9.	Unify	ml(N1,REST)	ml(1,[1])	SD_1/REST
10.	POP.			SD_3/REST
11.	POP.			SD_4/L
12.	END.			L = [3,2,1]

IV. SUMMARY AND CONCLUSIONS

The Prolog language has been proven to be useful for many applications. This study has added to the interpreter P5, the facility for the list data structure operation.

The interpreter is a good tool for the investigation of possible architectures and the understanding of the Prolog processing environment. The data types are extensively used by the interpreter. They provide means for memory organization and error checking.

The proposed data structure facility for P5 was implemented and tested. The example program makelist demonstrates the processing involved in the run-time generation of new list data structures.

The design decisions involving the choice of the data structure representation and implementation were presented. The choice for data structure implementation involved many trade-offs in terms of ease of representation and efficiency. The following decisions were adopted for list structure representation in P5. The source code list data structures are to be represented as contiguous words of memory (strings) and terminated by the constant NIL.

The run-time generated data structures use the Lisp cons cell, where the car can be an element of type constant or variable, and the cdr can be of type Structure_Data or the constant NIL.

The cons cell scheme used for run time structures adds flexibility to the interpreter in terms of the type of structures that can be represented but it introduces a cost in terms of dereferencing. The new data structure memory added to P5 can be updated at run time.

It should be noted that DM can grow very large since each activation of a procedure which creates lists adds to DM. This can be costly, especially in the case of recursive procedures. For this reason the cons cell format and indirection through a table is adopted. The table contains maximum size pointers; double word pointers can also be considered. This is especially important in P5 since run-time memory management is not specified.

The C programming language was used to write the interpreter because of its functionality and low level handling capabilities. The interpreter was compiled and executed under Lattice C and Aztek C compilers. It should be easily portable to other machines.

A Host system, XLISP, will be added to provide a suitable environment for development. With this capability the interpreter can be further tested and improved.

Further efforts should be centered on developing the front-end part of Prolog system. Some of these are the compiler of Prolog source strings to interpreter code strings used by P5 and the development of Built-Ins and I/O facilities.

APPENDIX A

SAMPLE RUN OF MAKELIST EXAMPLE

The initialized memory structures are included in Appendix B.

```
MAIN   gsp= 23  ge=c012
FETCH
UNIFY  ge=   3  re=   1  rf=   0
UNDO   fliste=7006
PUSH
UNIFY  ge=   3  re=400a  rf=   1
BIND.  C=    3  V= 400a  G =  24   F=   3
UNIFY  ge=400f  re=d002  rf=   1
BIND.  C= 400f  V= d002  G =  24   F=   3

MAIN   gsp= 8   ge=c022
FETCH
UNIFY  ge=400a  re=   3  rf=   0
UNIFY  ge=400b  re=   2  rf=   0
BIND.  C=    2  V= 400b  G =   9   F=   5

MAIN   gsp=11   ge=c012
FETCH
UNIFY  ge=400b  re=   1  rf=   0
UNDO   fliste=7006
PUSH
UNIFY  ge=400b  re=400a  rf=   1
BIND.  C=    2  V= 400a  G =  12   F=   3
UNIFY  ge=400e  re=d002  rf=   1
BIND.  C= 400e  V= d002  G =  12   F=   3

MAIN   gsp= 8   ge=c022
FETCH
UNIFY  ge=400a  re=   3  rf=   0
UNDO   fliste=6014
UNIFY  ge=400a  re=   2  rf=   0
UNIFY  ge=400b  re=   1  rf=   0
BIND.  C=    1  V= 400b  G =   9   F=   6

MAIN   gsp=11   ge=c012
FETCH
UNIFY  ge=400b  re=   1  rf=   0
UNIFY  ge=400e  re=d001  rf=   0
BIND.  C= d001  V =400e  G =  12   F=   2
```

```
MAIN   gsp=14   ge=1000
POP

MAIN   gsp=14   ge=1000
POP
END OF RUN

           CONTENTS OF A MEMORY

   C Field   V Field  G Field  F Field
NIL                                      gsre= 26 arl=  0
     4005      4001       1a        0
NIL                                      gsre= 26 arl=  3
        3      400a       18        3
     400f      d002       18        3
        2      400b        9        5
NIL                                      gsre= 14 arl= 10
        2      400a        c        3
     400e      d002        c        3
        1      400b        9        6
     d001      400e        c        2
NIL                                      gsre= 26 arl=  3
        3      400a       18        3
     400f      d002       18        3
        2      400b        9        5
     d003      400e     701e     7027
NIL                                      gsre= 26 arl=  0
     4005      4001       1a        0
     d004      400f     7031     703a

Contents of Data structure Memory:

      dm[1] =    1
      dm[2] = 1000
      dm[3] = 400a
      dm[4] = 400e
      dm[5] = 1000
      dm[6] =    2
      dm[7] = d001
      dm[8] = 1000
      dm[9] =    3
      dm[10]= d003
      dm[11]= 1000
```

APPENDIX B

INTERPRETER HEADER FILE

```
/*FILE NAME MLH.H*/
#define  type(x)     ( (x)>>12 )
#define  value(x)    ( (x) & 0X0FFF )
#define  gete(x,y)   ((x)(y))

#define  MASKT   0XF000     /*  mask type                  */
#define  MASKV   0X0FFF     /*  mask value                 */
#define  MASKRK  0X0FF6     /*                             */
#define  EOS     0X2000     /*  end of string element      */
#define  MASKF   0X00FF     /*                             */
#define  SD      0X000D     /*  structure data type        */
#define  SC      0X000C     /*  structure clause type      */
#define  RV      0X0004     /*  reference variable type    */
#define  NIL     0X0001     /*  NIL type                   */
#define  RPF     0X0006     /*  reference ptr fact type    */
#define  RPR     0X0007     /*  reference ptr rule type    */
#define  SL      0X000F     /*  structure list type        */
#define  C       0X0000     /*  constant type              */
#define  RC      0X7000     /*                             */


/*  symbol table variables / functors      */
/*  ml   = 1     FUNCTOR                   */
/*  dec = 2      FUNCTOR                   */
/*  N    = A     RV                        */
/*  N1   = B                               */
/*  REST= E                                */
/*  L    = F                               */
/*  SD1= 1                                 */
/*  SD2= [N | REST]                        */
/*                                         */
/*   ml( 1, 1 ).                           */
/*   ml( N,[N | REST]) :- dec(N,N1) , ml( N1, REST).   */
/*   goal ml(3,L) ?                                    */

unsigned int  m[50] = {
       0X0000,  /* 0:                        */
       0X0000,  /* 1: ml(    fact            */
       0X0001,  /* 2: 1, constant = 1        */
       0XD001,  /* 3: 1)   SD1               */
       0X0000,  /* 4: . nil                  */
```

```
        0X0000,  /* 5: ml(    rule        */
        0X400A,  /* 6: N          RVA     */
        0XD002,  /* 7: N1,REST  SD2       */
        0XC022,  /* 8: dec(               */
        0X400A,  /* 9: N       RV         */
        0X400B,  /*10: N1      RV         */
        0XC012,  /*11: ml(                */
        0X400B,  /*12: N1,     RV         */
        0X400E,  /*13: REST)   RV         */
        0X1000,  /*14: . nil              */
        0X0000,  /*15: DEC(               */
        0X0003,  /*16: 3,   constant      */
        0X0002,  /*17: 2 )  constant      */
        0X0000,  /*18: . nil              */
        0X0000,  /*19: DEC(               */
        0X0002,  /*20: 2,   constant      */
        0X0001,  /*21: 1)   constant      */
        0X0000,  /*22: . nil              */
        0XC012,  /*23: ml(  SC arity 2    */
        0X0003,  /*24: 3,   constant      */
        0X400F,  /*25: L)   RV call       */
        0X2000   /*26: EOS END OF QUERY   */
      };

unsigned int stb[10] ={

        /* STRUCTURE TABLE                          */
        0X1000,      /* 0:C,NIL                     */
        0XF001,      /* 1:SL,1  LIST OF ml          */
        0XF004,      /* 2:SL,4  list of dec         */
        0X1000       /* 3:C,NIL                     */
      };

unsigned int flist[10] ={

        /* FLIST */
        0X1000,      /* 0:C,NILL     */
        0x6002,      /* 1:RPF, ml    */
        0X7006,      /* 2:RPR, ml    */
        0X2000,      /* 3:E0S        */
        0X6010,      /* 4:RPF, DEC   */
        0X6014,      /* 5:RPF, DEC   */
        0X2000       /* 6:E0S        */
      };

unsigned int  a[150] ={

        /*STATE MEMORY A                        */

        0X1000,        /*0:C,NIL                */
```

```
        0X001A,         /*1:gsre=26(EOS)       */
        0X0000,         /*2:arl=0              */
        0x4005,         /*3:C=RV:5             */
        0X4001,         /*4:V=RV:1(PASSED VAR)*/
        0X001A,         /*5:G=26(EOS)          */
        0X0000,         /*6:F=NULL             */
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        };

unsigned int dm[40] = {
        /* data space memory for lists*/
0X1000,  /*0 CONSTANT  NIL              */
0X0001,  /*1 CONSTANT  1       SD1      */
0X1000,  /*2 CONSTANT  NIL              */
0X400A,  /*3 RV N       a      SD2      */
0X400E,  /*4 RV REST    e               */
0X1000   /*5 NIL                        */
                    };


unsigned int dstb[10]= {
        /*  data structure lookup table */
0X1000,  /*0 CONSTANT NIL                */
0X5001,  /*1 REFERENCE POINTER  SD1      */
0X5003,  /*2 REFERENCE POINTER  SD2      */
0X1000   /*3 CONSTANT NIL                */
       };

unsigned int  dgev = 0;
unsigned int  drev = 0;
unsigned int  dgsp  = 0;
unsigned int  drsp  = 0;
unsigned int  duf   = 0;
unsigned int  dgspv  = 0;
unsigned int  drspv  = 0;

unsigned int  gsh = 0;   /*  pointer to goal string head  */
unsigned int  gsp = 23;  /*  goal string pointer          */
unsigned int  rsp = 0;   /*  rule string pointer          */
unsigned int  rf  = 0;   /*  rule = 1 fact = 0            */
unsigned int  arity = 0; /*  # of arguments               */
unsigned int  ge = 0;    /*  goal element                 */
unsigned int  get = 0;   /*  goal element type            */
unsigned int  gev = 0;   /*  goal element value           */
unsigned int  re = 0;    /*  rule element                 */
unsigned int  ret = 0;   /*  rule element type            */
unsigned int  rev =0;    /*  rule element value           */
unsigned int  stbe = 0;  /*  structure table element      */
unsigned int  stbet = 0; /*  structure table element type */
```

```
unsigned int  stbev = 0; /*  structure table element value*/
unsigned int  fliste = 0;/*  flist element                 */
unsigned int  flistet= 0;/* flist element type            */
unsigned int  flistev= 0;/* flist element value           */
unsigned int  tap = 0;   /*  temporary a pointer          */
unsigned int  ap = 7;    /*  a memory pointer             */
unsigned int  arc = 3;   /*  activ record pointer         */
unsigned int  frm = 0;   /*  fact rule marker             */
unsigned int  functor=0; /*  functor                      */
unsigned int  uf = 0;    /*  unify flag 0 or 1            */
unsigned int  found = 0; /*  fetch flag                   */
unsigned int  bindcnt=0; /*  # of bindings                */
unsigned int  endlist=0; /*                               */
unsigned int  i =0;      /*                               */
unsigned int  gkey = 0;  /* goal key                      */
unsigned int  rkey = 0;  /* rule key                      */
unsigned int  fkey = 0;  /* fact key                      */
unsigned int  tge = 0;   /* temporary goal element        */
unsigned int  tta = 0;   /*                               */
unsigned int  ta[150];   /* temporary a memory            */
unsigned int  tx[100];   /*                               */
unsigned int  x=0;       /*                               */
unsigned int  iap=0;     /*                               */
unsigned int key=0;      /*                               */
unsigned int dstbend = 3;/* end of dstb marker            */
unsigned int dmend   = 6;/* end of dm   marker            */
unsigned int newds   = 0;/* new data structure variable   */
unsigned int buffer[20]; /* buffer area for new SD        */
unsigned int ddge = 0;
unsigned int ddgev = 0;
unsigned int ddget = 0;
```

INTERPRETER MODULE MAIN()

```
#include "mlh.h"  /* HEADER FILE WITH INITIALIZED MEMORIES*/
#include "mldp.c" /* DATA STRUCTURE UNIFY MODULE          */
#include "mlf.c"  /* FETCH MODULE                         */
#include "mlu.c"  /* UNIFY MODULE                         */

main ( )
{

while(( ge = gete(m,gsp++)) != EOS )
    {
     get = type(ge); gev = (ge & MASKV) ;
printf("MAIN  gsp=%2d ge=%x \n",gsp-1,ge);
     switch( get )
         {
          case SC: found = fetch( );
                   if(found == 1)
                       {if(rf == 1) push( );
                        uf=unify( );
                       }
                    break;
          case NIL: pop( );
                    break;
          default:  printf(" get error =%x\n",get);
                    break;
          }


/* printout of state of Prolog machine */
printf(" C Field  V Field  G Field  F Field\n");
for ( i=0; i < ap ;  ) {
 if(ai == 0X1000)  { printf("NIL");
printf(" gsre=%3d arl=%3d \n",ai+1,ai+2); i+=3;
       }
printf(" %4x  %4x  %4x  %4x\n"
 ,ai,ai+1,ai+2,ai+3); i+=4;
         }
     if (found != 1)
         { printf("FETCH ERROR:functor=%x
frm=%x\n",functor,frm);
           break;
         }
```

```
     if (uf == 0 )
         { printf("UNIFY ERROR:gsp=%d rsp=%d arc=%d
ap=%d\n",gsp,rsp,arc,ap);
           break;
         }
     if (ge==0)
  { printf("Error with ge\n");
    break;
  }
      }/* end of while */
printf("END OF RUN\n");
printf(" C Field  V Field  G Field  F Field\n");
for ( i=0; i < ap ;  ) {
 if(ai == 0X1000)  { printf("NIL");
 printf(" gsre=%3d arl=%3d \n",ai+1,ai+2); i+=3;
       }
printf(" %4x  %4x  %4x  %4x\n"
 ,ai,ai+1,ai+2,ai+3); i+=4;
         }
printf("Data structure Memory\n");
for (i=1;i < dmend; i++) printf(" dm%2d = %4x \n",i,dmi);

}    /* END OF MAIN */
```

INTERPRETER MODULE PUSH()

```
                        push( )
                           {
 aap++=0X1000; /* NIL for end of curr. function activation
record (arc)*/
   aap++=gsp+arity; /* goal string re-entry point (gsre)*/
 aap++=arc;       /* back link to arl; arc becomes arl after
push */
 arc=ap;            /* new arc body pointer;gsre and arl
pointers are head*/
                     printf("PUSH\n");
                          return;
                             }
```

INTERPRETER MODULE POP()

```
/* START OF POP */

pop( )
{
unsigned  int temp,tarc,i;
   tta=0;x=0;iap=arc;rf=0;
   aap++=0X1000;
   gsp=aarc-2;
   tarc=ap+2;

   printf("POP\n");

   while( iap < ap-2 )  /* buildta while loop */
        {
          temp=aiap; temp=temp>>12;/*get next Cfield
element*/

           if( temp  == RV )/*test for calling variable*/
                {
                  key=aiap+1;      /*get Vfield element*/
/*check for data structure*/
  if(type(key) == 0X000d){ buildata(key,iap); goto here; }

                  findbv( );       /*use findbv to get ap of
matching Vfield*/

                  printf("POP3. tap=%d iap=%d\n",tap,iap);

                  if(tap != 0) buildta(iap);/*build RC,tx
for copyta*/
                 }/*close of sucessful test*/
here:           iap=iap+4;
         }/*repeat for next arc arecord*/
/* end of buildta */

/* start of copy arl to new arc */

iap=(a(arc-1)-2);
while(aiap != 0x1000) /* debug  */
    {
     aap++=aiap;
     iap++;

     }
/* end of copy */
```

```
/* start of copy of returned constants (RC's) to new arc */

if(tta !=0)
    {
      copyta( );
     }/*copy RC's*/

arc=tarc;

i=arc-2;
 return;
/* END OF POP */
}



bind( )

{
 printf("BIND. ");
 aap++=re;
 aap++=ge;
 aap++=gsh;
 aap++ = frm;
 printf(" C=%4x   V=%4x   G=%4d   F=%4d
\n",aap-4,aap-3,aap-2,aap-1);
 return;
}
```

```
buildta(iap)
int iap;
{
   if(tta == 0) x=0;/*initialize sort pointer*/
   txx++=tap-1;txx++=tta;/*build tx*/
   tatta++=atap-1;     /*C in Cfield*/
   tatta++=aiap;       /*V  " V  "   */
   tatta++=(iap+1 | RC); /*apV in Gfield*/
   tatta++=(tap+1 | RC); /*apC in Ffield*/
   return;
}

copyta( )
{
   int intchgf,temp,i,x;
   if (tta/4 > 1 )/* if  more than one RCrecord then sort */

      {
        for( i=0; i=tta/4-1;i++) /*sort tx */
             {
               intchgf=0;
               for ( x=0; x=tta/2-1; x+=2)
                   {
                     if ( txx > txx+2 )
                        {
                          temp=txx;txx=txx+2;txx+2=temp;
                          temp=txx+1;txx+1=txx+3;txx+3=temp;
                          intchgf=1;
                        }
                    }


               if(intchgf == 0) break;/*exit ifor loop*/
             }
        for(i=0; i=tta/2 ;i+=2) /* copy ta to a */
             {
               for( tta=txi+1; tta=txi+1+4; tta++)
                    aap++=tatta++;

             }

        return;
      }
/* if sort isn't needed then copy ta to a */
   for(tta=0;tta <= 3; tta++) aap++=tatta;
   return;
/* END OF COPYTA */
}
```

```
buildata(key1,iapoint)
unsigned int key1, iapoint;
{
int jix=1;
int kk = 1;
dgev = value(key1);
dgsp = dstbdgev;
dgspv = value(dgsp);
while(( ddge = dmdgspv++) != 0X1000) {
 ddget=type(ddge);
 ddgev=value(ddge);
  if (ddget == RV) { key = ddge; findbv();
    if( tap != 0) bufferjix = atap-1;
    if( tap == 0) bufferjix = ddge;
    }
  if (ddget != RV)  bufferjix = ddge;
  jix = jix + 1;
         } /*end of while */
/* create new data structure with new values of variables */
newds = 0xD000 | dstbend; /*  generate SD(dstbend) D003*/
dstbdstbend = newds;
dstbend++;
for ( kk=1; kk < 3; kk++){
 dmdmend++ = bufferkk;}
 dmdmend++ = 0X1000; /* add the nil */
buildnewta(iapoint);
}

buildnewta(iap)
unsigned int iap;
{
   tatta++=newds;        /*C  New structure Data   */
   tatta++=aiap;       /*V  returned Variable    */
   tatta++=(iap+1 | RC); /*apV in Gfield         */
   tatta++=(tap+1 | RC); /*apC in Ffield         */

   return;
}
```

INTERPRETER MODULE FETCH()

```
fetch( )

{
       extern unsigned int  functor, stbe, stbet, stbev;
       extern unsigned int  fliste, flistet, flistev;
       extern unsigned int stb;
       extern unsigned int flist;
       extern unsigned int gev,arity,rf,rsp,frm;

arity =( gev & 0X000F) ;
functor = gev; functor= functor>>4;
printf("FETCH  \n");

if ( (stbe= stbfunctor) == 0X2000 ) return(0);
    stbet = stbe>>12;stbev = (stbe & 0X0FFF);
    switch ( stbet )
         {
          case 6 :frm = 0;rsp = stbev;rf = 0;
                     break;
          case 7 :frm = 0;rsp = stbev;rf = 1;
                     break;
          case 15 :frm = stbev;
                   if(( fliste = flistfrm++) == 0X2000 )
printf("perror(3)");
                   flistet =fliste>>12;flistev=(fliste &
0X0FFF) ;

                   switch(flistet)
                          {
                           case 6 :rsp = flistev;rf = 0;
                                    break;
                           case 7 :rsp = flistev;rf = 1;
                                    break;
                           default:printf("perror(4)");
break;
                          } break;
         default:printf("perror(5)");break;
         }
return(1);
}
```

INTERPRETER MODULE UNIFY()

```
unify( )
{
unsigned int rfkey;
unsigned int dflag;
dflag = 0;
gsh =gsp;bindcnt = 0;
i = 1;endlist = 0;
    while( endlist == 0 )
    {
    if (i >  arity ) break;
    i++;
    ge = gete(m,gsp); gsp++;
    re = gete(m,rsp); rsp++;
    get = type(ge);gev = value(ge);
    ret = type(re);rev = value(re);

    printf("UNIFY  ge=%4x  re=%4x  rf=%d\n",ge,re,rf);

    if(get == RV) {  key=ge; findbv( ); }
    gkey=get & 0X000C; rkey=ret >> 2 ;
    rfkey = rf << 4;
    fkey= gkey | rkey | rfkey ;
    if(rf != 0){tge = ge ;ge = re;re = tge;}
        switch(fkey) /* fkey= rf(R or F),gtype(C or
V),rtype(C or V) */
             {
                 /* case FCC*/
               case  0: if(ge != re){gsp = gsh; endlist =
undo( );
                                    i=1;gsp = gsh;bindcnt=0;
                                  } break;  /* case FCC */
                 /* case RCC */
               case 16: if(ge != re){ gsp=gsh; endlist =
undo( );
                                    i=1;gsp = gsh;bindcnt=0;
                                  } break;   /* case RCC */
               case  4 : bind( );bindcnt++;break;
               /* case FVC */
               case 17 : bind( );bindcnt++;break;
               /* case RCV */
               case 21 : bind( );bindcnt++;break;
               /* case RVV */
 /* here start structure data cases */
 /* case FDD: Data structure with Data struc  under Fact */
```

```
                case 15 : dflag = dp(ge,re);if(dflag ==
0){endlist = undo();
       printf("dp failed\n");
       i=1;gsp = gsh;bindcnt=0;
      } break;
/* case RDD: Data structure with Data struc  under Rule*/
                case 31 : dflag = dp(ge,re);if(dflag ==
0){endlist = undo();
      printf("dp failed\n");
      i=1;gsp = gsh;bindcnt=0;
      } break;
/* case RDV: Data structure with a Variable  under Fact*/
                case 29 : bind(); bindcnt++; break;
/* case FDV: Data structure with a Variable  under Rule*/
                case 13 : bind(); bindcnt++; break;
/* case RVD: Variable with a Data structure  under Rule*/
                case 23 : bind(); bindcnt++; break;
/* case FVD: Variable with a Data structure  under Fact*/
                case  7 : bind(); bindcnt++; break;
              default :printf("Error fkey=%d\n",fkey);break;
             }
     }
if(endlist == 1) return(0);
if(rf != 0){gsp = rsp;rf = 0;}
return(1);
}
findbv( )   /* search for bound variable */
{
unsigned int temp;
if( rf == 0 ) tap=arc+1; else tap=aarc-1+1;
if ( tap == 1 ) return; /* start of a */;
while( tap < ap ) /*test for end of arc record*/
       {
if (atap-1 == 0X1000 ) break; /*test for end of arl record*/
        if (atap == key )/*test for V-field entry = key*/
          {
                tge = atap-1;/* if Cfield entry is a C then
replace ge*/
            temp=tge>>12; /* and return. tap is at Vfield
for return*/
            if( temp != 4 )
                  {  /*from inside while*/
                     ge=tge; get=type(tge); gev=tge&0X0FFF;
                     return;
                   }
          }
        tap+=4;       }
tap=0;
return;
}
```

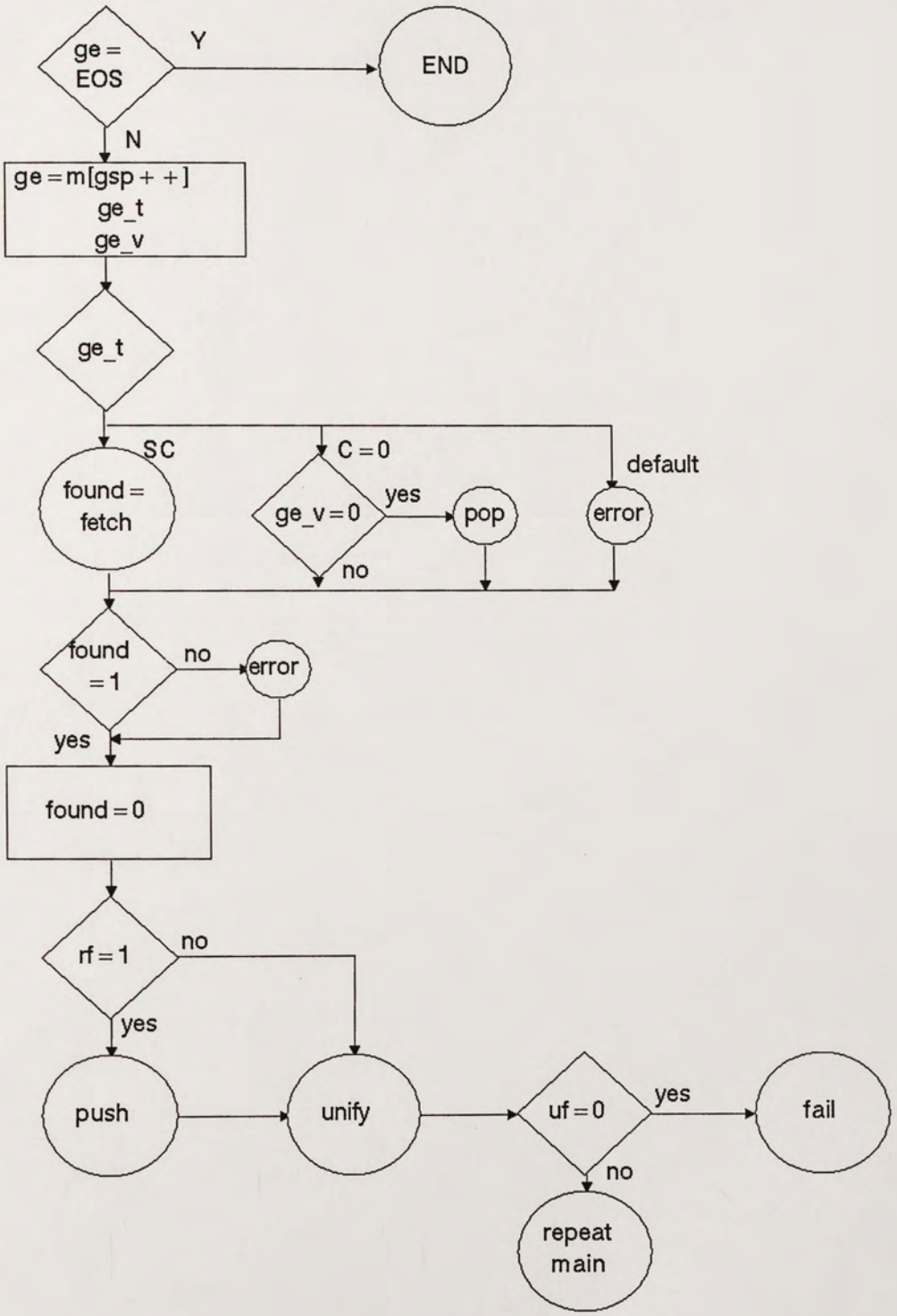
```
undo( )
{

   if (rf == 1)printf("perror(200)");
   printf("UNDO  fliste=%x \n",flistfrm);
   if((frm == 0) || (fliste = flistfrm) == EOS)
   return(1);
   ap=ap-(4*bindcnt);
   fliste=flistfrm++;
   flistet=fliste>>12;flistev=fliste&0X0FFF;

   switch(flistet)
   {case 6 :rsp=flistev;rf=0;break;
    case 7 :rsp=flistev;rf=1;push( );
 break;
    default:printf("flistet error=%x\n",flistet);return(1);
break;
   }
   return(0);
}
```

INTERPRETER MODULE DP()

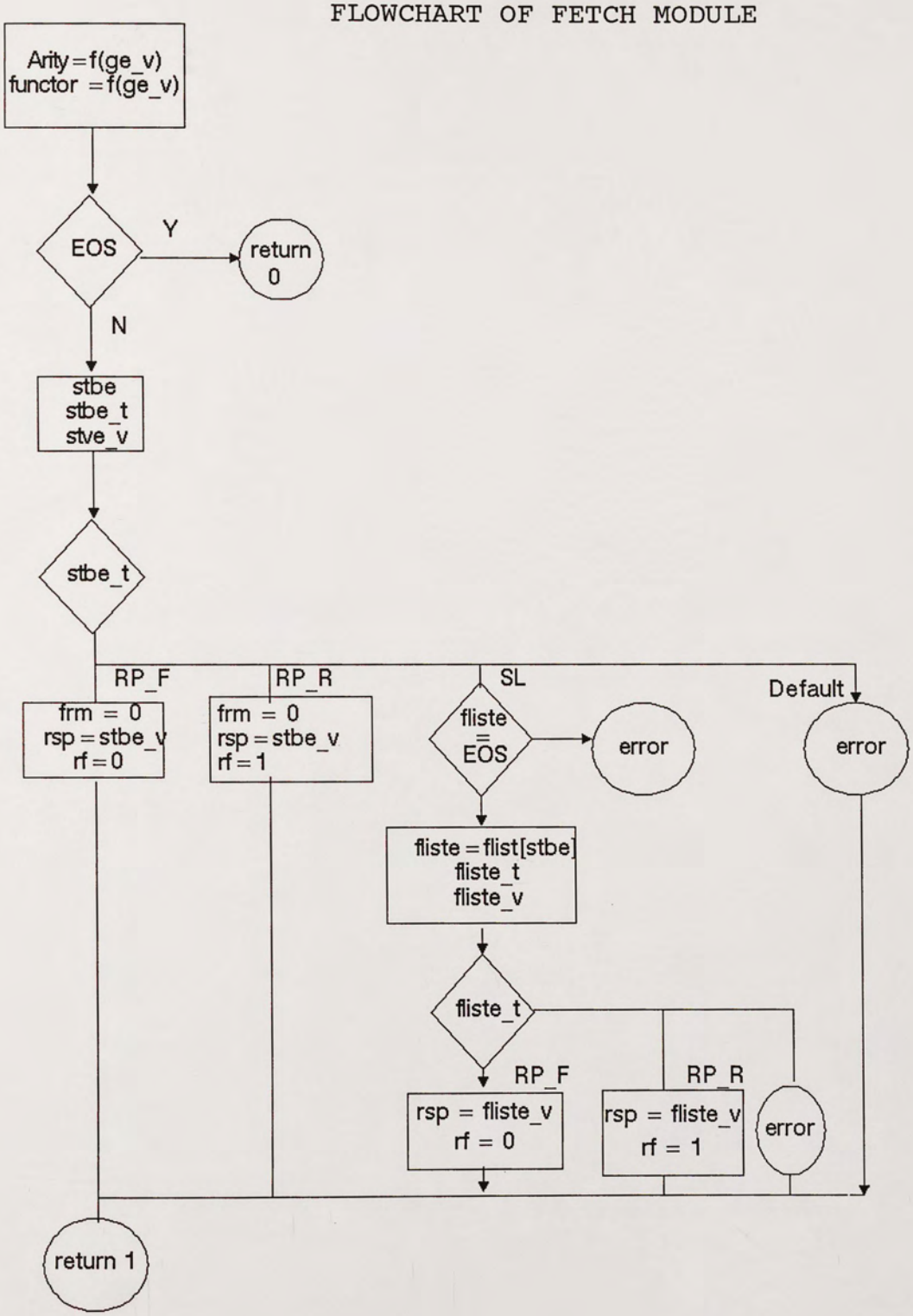
```
/* subprogram name dp.c */
/* this will be called when in unify (structure data,
structure data) */

dp(dge,dre)
unsigned int dge,dre;
{

dgev = value(dge);
drev = value(dre);
dgsp = dstbdgev;
drsp = dstbdrev;
        duf = du(dgsp,drsp);
        if (duf == 1) return(1);
        if (duf == 0){ printf("dp error\n"); return(0);}
}

/* start of du.c */

du(dgsp,drsp)
unsigned int dgsp,drsp;
{
unsigned int flag;
unsigned int rfkey;

dgspv = value(dgsp);
drspv = value(drsp);
 flag = 0;
/* this will loop thru all elements of list until NIL is
reached */
while ( (dmdgspv != 0X1000) || (dmdrspv != 0X1000) )
  {
        ge=dmdgspv;
        re=dmdrspv;
        dgspv++;
        drspv++;

    printf("DU1.Goal element=%4x   Rule element=%4x
rf=%d\n",ge,re,rf);

    get = type(ge);gev = value(ge);
    ret = type(re);rev = value(re);

    if(get == RV) {  key = ge; findbv( ); }
```

```
    gkey = get & 0X000C; rkey=ret >> 2 ;
    rfkey = rf << 4;
    fkey= gkey | rkey | rfkey ;
     if(rf != 0){tge = ge ;ge = re;re = tge;}


        switch(fkey) /* fkey= rf(R or F),gtype(C or
V),rtype(C or V) */
             {

                case  0: if(ge != re){ flag = 1;  /* case
FCC*/
      printf("Contants do not unify\n");
                                        }  break;
                case  4 : bind( );bindcnt++; break;      /*
case FVC */
                case 21 : bind( );bindcnt++; break;      /*
case RVV */
                case 17 : bind( );bindcnt++; break;      /*
case RCV */
               default :printf("error in DU
fkey=%d\n",fkey);break;
             } /* end of switch */
   } /* end of loop */


  if(flag == 1) return(0);/* which one comes first? */
  if(rf != 0){gsp = rsp;rf = 0;}
 return(1);
} /* end of function du */
```

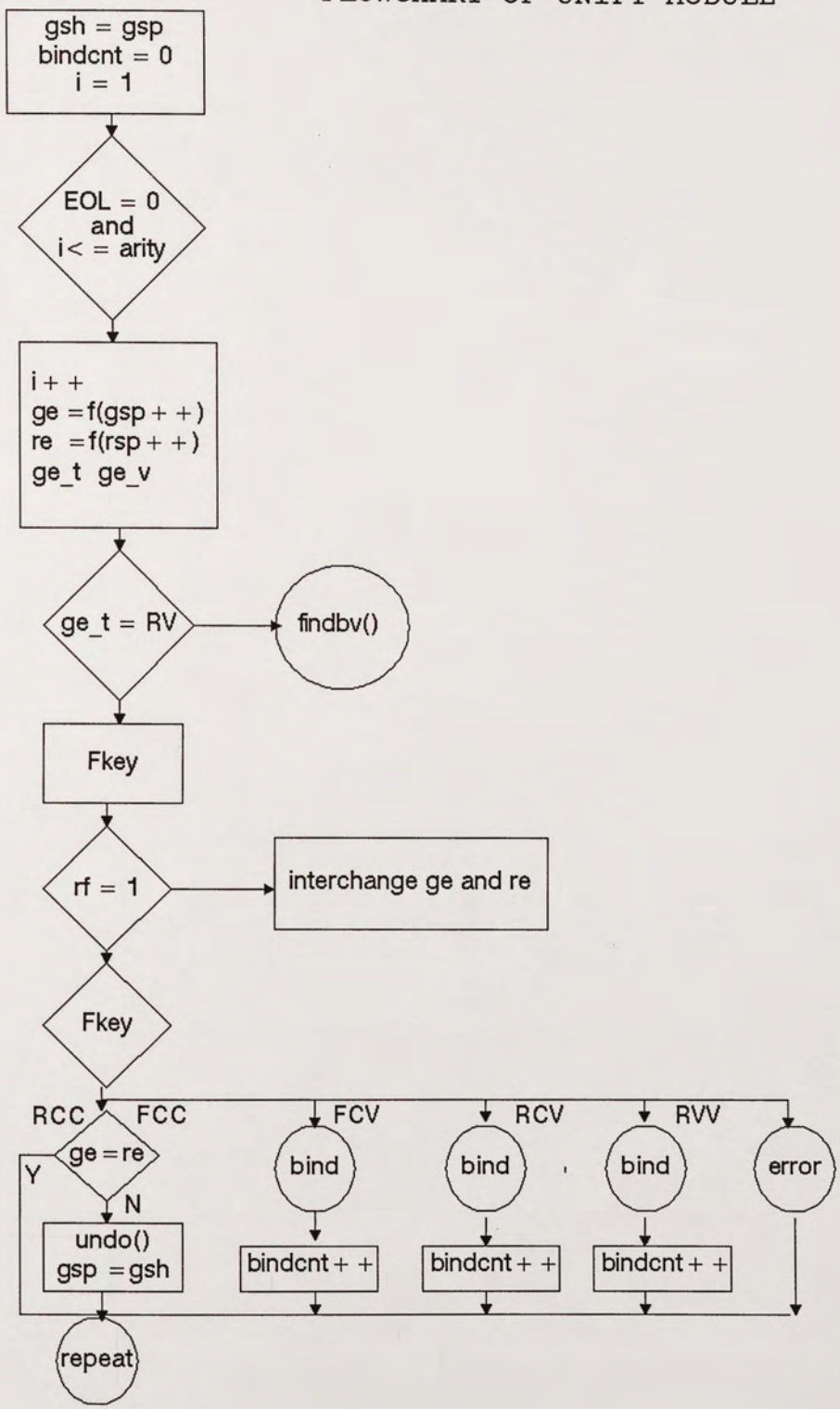
APPENDIX C
FLOWCHART OF MAIN MODULE



FLOWCHART OF FETCH MODULE



FLOWCHART OF UNIFY MODULE



LIST OF REFERENCES

{1} Warren, David H. Prolog-The Language and its Implementation Compared with Lisp.Edinburgh, Scotland: University of Edinburgh, 1977.

{2} Degroot, Doug. Prolog and Knowledge Information Processing: A Tutorial. New York: IBM Research, T.J. Watson Research Center, 1984.

{3} Despain, Alvin M. and Patt, Yale N. "Aquarius--A High Performance Computing System for Symbolic/Numeric Applications, IEEE Proceedings. October, 1984.

{4} Campbell, J. A. Implementations of Prolog. New York: Ellis Horwood Limited, 1984.

{5} Clocksin, W.F. and Mellish, C.S. Programming in Prolog. New York Springer-Verlag, 1981.

{6} Dobry, T.P., Patt, Y.N., and Despain, A.M. "Design Decisions Influencing the Microarchitecture for a Prolog Machine," Micro 17 Proceedings. October, 1984.

{7} Steele, Guy Jr. Common Lisp: The Language. Burlington, Massachussets: Digital Press, 1984.

{8} Turbo Prolog-The Natural Language of Artificial Intelligence. Boreland International, 1986.

{9} Despain, Alvin M. and Chang, Jung-Herng "AND-Parallelism of Logic Programs Based on A Static Data Dependency Analysis," IEEE Proceedings. February, 1985.

{10} Jackson, Peter. Introduction to Expert Systems. Addison-Wesley, 1985.