

IMPLEMENTATION OF PROLOG

S. KESHAV &
INDERPAL SINGH MUMICKI

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, DELHI
MAY, 1986

IMPLEMENTATION OF PROLOG

by

S. Keshav and Inderpal Singh Mumick

Supervisor : Dr. Niraj Sharma

MAY 1986

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

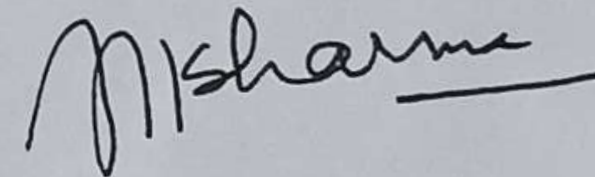
INDIAN INSTITUTE OF TECHNOLOGY

DELHI

CERTIFICATE

This is to certify that the report entitled "Implementation of Prolog" is being submitted by S. Keshav and Inderpal Singh Mumick to the Indian Institute of Technology, Delhi in part fulfillment of requirements for the degree of Bachelor of Technology in Computer Science and Engineering. This report is a record of the work carried out under my supervision and has not been submitted to any other University or Institute towards any degree.

Date : May 8, 1986



Dr. Niraj Sharma

Department of
Computer Science and
Engineering, IIT Delhi.

ACKNOWLEDGEMENT

We would like to gratefully acknowledge support and help from our project supervisor Dr. Niraj Sharma, as also from Prof. P. C. P. Bhatt , who suggested this project and keenly guided our literature survey, and Mr. N. C. Kalra, who kept the OMC 58000 system available to us against all odds.

The excellent diagrams in this report have been drawn by Miss Suchitra Srinivasan.

Inderpal Singh Mumick

S. Keshav

ABSTRACT

Prolog is a powerful logic programming language developed at the University of Marseilles, France around 1970. The language is an implementation of First Order Predicate Logic. Its main attraction is the ease of programming. Clear, readable, concise and error free programs can be quickly developed.

This report describes an implementation of Prolog that we have designed for the OMC 58000 running under the OMNX operating system. We have written an interpreter, and have provided many of its predefined predicates. Almost the entire language as described in "Programming in Prolog" by Clocksin and Mellish has been implemented. The interpreter is based on the concepts of structure sharing, unification, and the lush control procedure as described by Warren, Kowalski and Sammut.

CONTENTS

<u>ONE</u>	<u>INTRODUCTION</u>	... 1
	1.1 Application Areas, 1	
	1.2 History of Prolog Implementation, 1	
	1.3 Data Structures, 2	
	1.4 Prolog Procedures, 2	
	1.5 The Logical Variable, 3	
	1.6 Dual Semantics of Prolog, 3	
<u>TWO</u>	<u>THE PROLOG LANGUAGE : A BRIEF DEFINITION</u>	... 5
	2.1 Syntax and Terminology, 5	
	2.2 Declarative and Procedural Semantics, 7	
	2.3 The Cut Operation, 8	
<u>THREE</u>	<u>AN OVERVIEW OF THE PROLOG IMPLEMENTATION</u>	... 10
<u>FOUR</u>	<u>THE LEXICAL ANALYSER</u>	... 12
	4.1 The Token Buffer, 12	
	4.2 Rules for Token Recognition, 13	
	4.3 Program Outline for the Lexical Analyser, 13	
<u>FIVE</u>	<u>THE PROLOG DATABASE</u>	... 15
	5.1 Atom Record, 15	
	5.2 Integer Record, 16	
	5.3 Variable Record, 17	
	5.4 Functor Record, 18	
	5.5 Clause Record, 20	
	5.6 Termlist Record, 22	
	5.7 Goaltree Record, 22	
	5.8 Goallist Record, 23	
	5.9 The Procedure Table, 24	
	5.10 Example, 24	
<u>SIX</u>	<u>THE PARSER</u>	... 26
	6.1 The Grammar, 26	
	6.2 Clause Parser, 27	
	6.3 DATF Parser, 30	
	6.4 ATF and Term Parser, 32	
	6.5 Structure Parser, 36	
	6.6 List Parser, 37	

<u>SEVEN</u>	<u>UNIFICATION</u>	... 41
	7.1 Structure Sharing, 41	
	7.2 The Unification Algorithm, 45	
<u>EIGHT</u>	<u>CONTROLLING EXECUTION : LUSH</u>	... 54
	8.1 State of the Prolog System, 55	
	8.2 The Lush Algorithm, 56	
	8.3 Discussion of the Algorithm, 58	
	8.4 Example, 61	
	8.5 Interfacing of Predefined Predicates, 64	
<u>NINE</u>	<u>THE CUT OPERATION</u>	... 65
	9.1 Reset Backtrack Environment, 65	
	9.2 Shrink Trail Stack, 65	
	9.3 Recover Control Stack Space, 65	
<u>TEN</u>	<u>THE PREDEFINED PREDICATE INTERFACE</u>	... 66
	10.1 Implementation of some important Predefined Predicates, 66	
	10.2 How to Write your own Predefined Predicates, 69	
<u>ELEVEN</u>	<u>ERROR HANDLING</u>	... 73
<u>TWELVE</u>	<u>ORGANIZATION OF PROGRAM INTO FILES</u>	... 74
	12.1 Introduction, 74	
	12.2 Files, 74	
<u>THIRTEEN</u>	<u>PROBLEMS FACED DURING IMPLEMENTATION</u>	... 76
<u>FOURTEEN</u>	<u>USER MANUAL</u>	... 78
	14.1 The User Interface, 78	
	14.2 Using the System, 79	
	14.3 A Sample Session, 80	
	14.4 Modifications to Clocksin and Mellish, 81	
	14.5 Restrictions, 82	
	14.6 Limits of the System, 82	
	14.7 Available Predefined Predicates, 82	
<u>REFERENCES</u>		... 84

APPENDIX 1

... 85

APPENDIX 1A

... 86

APPENDIX 2

... 88

APPENDIX 3

... 89

CHAPTER ONE

INTRODUCTION

This report describes techniques for efficiently implementing the programming language Prolog. It is written mainly for those having some familiarity with Prolog. Those unfamiliar should first refer to Chapter 1 in the book "Programming in Prolog" by Clocksin and Mellish.

We begin with a brief discussion of the language. This is followed by a precise definition of the syntax and terminology in Chapter 2.

1.1 APPLICATION AREAS

Prolog is a simple but powerful programming language developed as a practical tool for logic programming. Prolog is especially suited to symbol manipulation applications such as natural language processing, compiler writing, symbolic equation solving, theorem proving and relational data bases. It is particularly apt for many AI applications.

1.2 HISTORY OF PROLOG IMPLEMENTATIONS

The first implementation of Prolog was an interpreter written in Algol-W by Phillip Roussel [1972]. This work led to better techniques for implementing the language, which were realised in a second interpreter, written in Fortran by Battani and Meloni [1973]. A notable feature of this design is the novel and elegant structure sharing technique for representing structured data built up during computation. This representation enables structured data to be created and discarded very rapidly, in comparison with the conventional literal representation based on a tree of linked records.

The first Prolog compiler was written by David Warren [1974] at the University of Edinburgh. The compiler was developed on the DEC 10 and translated directly into DEC 10 assembly language. It used the same fundamental design, including the structure sharing technique that was developed for the second Marseille interpreter. However the implementation was 15 to 20 fold faster, owing to compilation, and also because it was possible to capitalise on the DEC 10 architecture which is particularly favourable to the structure sharing technique.

Since then many implementations have come up around the

world. Each one provides new and different features, but the basic design strategy has remained the same.

1.3 DATA STRUCTURES

Data in Prolog is represented by generalized trees, constructed from records of various types. An unlimited number of different types of records may be used and they do not have to be separately declared. Records can have any number of fields. There are no type restrictions on the fields of a record.

The conventional way of manipulating structured data is through the predefined construct and select functions (as in Lisp). However data manipulation in Prolog is through a pattern matching algorithm provided in a process called unification. The ease in using Prolog comes partly from the fact that this data manipulation is done automatically by the Prolog system and is not to be specified by the programmer .

1.4 PROLOG PROCEDURES

Prolog is an exceptionally simple language for the user. A Prolog computation consists of little more than a sequence of pattern directed procedure invocations. Since the procedure call plays such a vital part , it is allowed a more flexible mechanism than in other languages. The special features are ...

- 1) When a procedure **returns** it can send back more than one output, just as (in the conventional way) it may have received more than one input.
- 2) Which arguments of the procedure are inputs and which will be output doesn't have to be determined in advance. It may in fact vary from one call to another. This allows procedures to be multi-purpose.
- 3) A procedure may return several times sending back alternative results. This allows procedures to be non-determinate. This is in contrast to the determinate procedures in a conventional language whose execution terminates for ever with the return of a value. The process of reactivating a Prolog procedure which has already returned a value is known as **backtracking**.
- 4) There is no distinction in Prolog between procedures and what would conventionally be regarded as tables or files of data. Program and data are mixed together in procedures and are accessed in the same way. Thus a general Prolog procedure comprises a mixture of data (in the form of facts) and rules for computing further data.

1.5 THE LOGICAL VARIABLE

A Prolog variable differs from a variable in a conventional programming language. It is not a name for a specific memory location and is not and can not be assigned values by the programmer. The Prolog variable may in fact remain undefined for any period of time, meaning that no value is assigned to it. This may happen even while the variable is in use in a procedure. Thus the machine oriented concepts of assignment and pointers are not an explicit part of Prolog. Due to this special nature and more flexible behaviour the Prolog variable is known as the logical variable. We may summarize its special properties as ...

- 1) The computational behaviour of Prolog is such that the programmer need not be concerned whether or not a variable has been given a value at a particular point in computation, or with what this value might be.
- 2) The logical variable can be associated with a value of any type.
- 3) A Prolog procedure may return as output an **incomplete** data structure containing logical variables whose values have not yet been specified. These variables can be later filled in by other procedures. This is achieved automatically in the course of unification, but has the same effect as explicit assignments to the fields of the data structure.
- 4) Two logical variables can be matched together, so that both will have exactly the same status at all times. These variables are said to share.

1.6 DUAL SEMANTICS OF PROLOG

Prolog has the unique property that a Prolog program can be interpreted declaratively as well as procedurally. This property has significantly affected the design of the language. It is also the real reason behind the ease of use of the language.

For most programming languages a program is simply a description of a process. The only way to understand the program and see whether it is correct is to run it - either on a machine with real data, or symbolically in the mind. Prolog programs can also be understood this way, and indeed this view is vital when considering efficiency. We say that Prolog, like other languages, has procedural semantics. It is this semantics which determines the sequence of states passed when executing the program.

However there is another way of looking at a Prolog program which does not involve any notion of execution. Here the program is interpreted declaratively, as a set of descriptive statements about a problem domain. From this standpoint the facts and rules

Introduction

of a Prolog program are nothing more than a convenient shorthand for ordinary natural language sentences. Each fact/rule is a statement which makes sense in isolation. It describes an object that is separate from the program or machine itself. The program is correct if each statement is true.

This natural declarative reading is possible because the procedural semantics of Prolog is governed by an additional **declarative** semantics, inherited straight from logic. The declarative semantics defines what facts can be inferred true from the facts and rules given in the Prolog program. This is regardless of how the program is executed to actually infer those facts. That will be the province of procedural semantics.

The knowledge that Prolog is based on a declarative semantics allows the programmer to initially ignore procedural details and concentrate on the declarative essentials of the algorithm. He can break up the program into small independently meaningful units. This inherent modularity also reduces the interfacing problems when several programmers are working on a project.

CHAPTER TWO

THE PROLOG LANGUAGE : A BRIEF DEFINITION

The basic Prolog language can be considered as being made up of two parts . The first part consists of a set of logical statements, of a form known as Horn clauses. These are a special subclass of general clauses which do not have any disjunctions.

The second part of Prolog consists of a very elementary control language. Through this control information the programmer determines how the Prolog system is to set about satisfying a sequence of goals. The control language consists merely of sequencing information, plus a cut primitive which restricts the system from considering unwanted alternatives for a goal sequence.

As discussed earlier there are two distinct ways to understand the meaning of a Prolog program, one declarative and one procedural. As far as the declarative reading is concerned one can ignore the control component of the program. The declarative reading is used to see that the program is correct. The procedural reading is necessary to see whether the program is efficient or indeed practical. It must take care of the control information.

We will now summarize the syntax of Prolog and briefly describe its semantics (both declarative and procedural).

2.1 SYNTAX AND TERMINOLOGY

A Prolog program is a sequence of clauses. Each clause comprises a head and a body. The body consists of a sequence of zero or more goals. For example the clause written ...

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

has likes(mary,X) as its head and likes(mary,Y) and likes(Y,X) as the goals making up its body. A clause with an empty body is called a fact or a unit clause. For example

```
likes( mary, john ).           ... is a fact
```

The head and goals of a clause are all examples of compound terms (except the cut goal which is represented by the single character '!'). Since the head and the goals either succeed or fail during execution, they are referred to as boolean terms.

In general a term is either an elementary term or a compound term. An elementary term is either a variable or a constant.

A variable is an identifier made up from letters, digits and the underline character '_', and beginning with a capital letter or the underline.

X, Constant, _, Answer, _3_blind_mice ... are variables

The variable consisting of a single underline character is called the anonymous variable. There may be any number of anonymous variables in a clause and they are treated as distinct variables. The scope of other variables extends to the clause in which they are named.

Constants are of two kinds - atoms and integers. Atoms are identifiers falling into one of the following three syntactic categories ...

1) A sequence of letters, digits, and underlines beginning with a lower case letter.

2) A sequence of symbols (non letter, non digit).

3) Any sequence of characters enclosed within quotes.

For example : a, prolog, #-?, =, ==, south_pole, 'south-pole' are all atoms.

Integers are numbers represented by a sequence of digits.

A compound term, also called a functor or an atomic formula comprises a principal term and a list of zero or more terms called arguments. The functor is characterised by its name, which is the name of its principal term, and its arity, i.e. the number of arguments. For example the functor written as :

college(iit, technical, india)

has the name college and arity 3. Its arguments are iit, technical, and india. The above notation wherein the principal term is followed by the arguments in parenthesis is the standard or canonical notation for functors. In addition functors of arity one and two may be declared as infix, prefix or postfix operators to allow a representation of the form

2 * 3, X = Y, not P, N factorial

instead of

*(2,3), =(X,Y), not(P), factorial(N)

The principal functor of a boolean term(head or goal) is called a predicate. The sequence of clauses whose heads all have the same predicate is called the procedure for that predicate.

2.2 DECLARATIVE AND PROCEDURAL SEMANTICS

Declarative Semantics

The key to understanding a Prolog program declaratively is to interpret each clause as a shorthand for a statement of natural language. A non unit clause :

P :- Q, R, S.

is interpreted as :

P if Q and R and S.

We also have to interpret each boolean term in the program as a simple statement. To do this we will also need to give a uniform interpretation to each functor in the program. For example consider the following set of clauses for appending and reversing lists :

```
append( [], L, L ).
append( [X|L1], L2, [X|L3] ) :- append( L1, L2, L3 ).

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

Note : | is used for the vertical bar.

The interpretations to be given are :

[]	:	The null list.
[X Y]	:	The list with head X and tail Y.
append(X,Y, Z)	:	list Y when appended to list X gives list Z.
reverse(X, Y)	:	List Y is the reverse of list X.

Considering each variable to be an arbitrary object, the interpretation of the four clauses is :

- 1) The empty list concatenated with L gives L.
- 2) The list whose head is X and tail is L1, on concatenation with list L2 gives the list whose head is X and tail is L3, if L1 concatenated with L2 gives L3.
- 3) The reverse of the empty list is the empty list.
- 4) The reverse of the list whose head is H and tail is T is the list L, if Z is the reverse of T, and Z concatenated with the list whose head is H and tail is empty is the list L.

The declarative semantics of Prolog defines the set of boolean terms which may be deduced to be true according to the program. A boolean term can be deduced to be true if it is the

head of some clause instance and each of the goals of that clause can also be deduced to be true. A clause instance is obtained by substituting, for each of zero or more of its variables, a new term for all occurrences of the variable. This recursive definition of truth governs the declarative semantics of Prolog.

Procedural Semantics

Procedural semantics describes the way the Prolog system executes a program to test the truth of a boolean term. The sequencing of clauses and the sequencing of goals within clauses plays a critical role in this algorithm, affecting the ability of the system to infer truth.

Procedurally the head of a clause is interpreted as a procedure entry point and a goal is interpreted as a procedure call. The way in which a goal is executed to test if some instance of it is true is defined as follows :

To satisfy goal P, the system searches for the procedure for P. Clauses in the procedure are scanned until the head of one matches or unifies with P. The unification process finds the most general common instance of the goal and the clause. If a match is found, the matching clause instance is activated by executing in turn, from left to right, each of the goals in its body. If at any time the system fails to find a match for a goal, it backtracks. During backtrack the system rejects the most recently activated clause, undoing any substitutions made by the match with the head of this clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal. Execution terminates successfully if there are no more goals waiting to be satisfied. In this case the system has found an instance of the original goal which is true. Execution terminates unsuccessfully when all choices for matching the original goal P have been rejected. Execution may also never terminate.

In general, backtracking can cause execution of a goal P to terminate successfully several times. The different instances of P obtained represent different solutions. In this way the procedure corresponding to P enumerates a set of solutions.

We say that a goal (or the corresponding procedure) has been executed determinately if its execution is complete and no more alternative solutions exist. This can be detected by the absence of any alternative clauses to match the goals invoked during the execution.

2.3 THE CUT OPERATION

Besides the sequencing of goals and clauses, Prolog provides one other very important facility for specifying control information. This is the cut goal, written as '!'. This goal

The Prolog Language

is not related to the logic of the program and should be ignored as far as the declarative semantics is concerned. Examples of its use are :

```
member( X, [X|_ ] ) :- ! .
member( X, [_|L ] ) :- member( X, L ) .
compile( S, C ) :- translate( S, C ), !, assemble( C ) .
```

The effect of the cut operator is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the goal which invoked the clause containing the cut. In other words the cut operation commits the system to all choices made since the parent goal was invoked. It renders determinate all computation performed since and including invocation of the parent goal, up until the cut.

Thus the second example above may be read declaratively as "C is a compilation of S if C is a translation of S and C is assembled" and procedurally as "In order to compile S, take the first translation of S you can find and assemble it". If the cut were not used here, the system might go on to consider other ways of translating S, which though correct may be unnecessary or unwanted.

The above two uses of cut do not contradict the declarative reading of the program. The only affect is to cause the system to ignore superfluous solutions to a goal. This is the commonest use of cut.

Cut can also be used in a way that a part of the program can be interpreted only procedurally. One of such uses is the **cut fail combination**. For example to say that an object X is ordinary if it is not exceptional, one may write :

```
ordinary( X ) :- exceptional( X ), !, fail .
ordinary( X ) .
```

The declarative reading of the second clause indicates that every X is ordinary. This is clearly not implied by the procedure. A better way to write the same procedure may be :

```
ordinary( X ) :- not( exceptional( X ) ) .
```


CHAPTER THREE

AN OVERVIEW OF THE PROLOG IMPLEMENTATION

An implementation of Prolog rests on the design of :

- 1) A lexical analyser.
- 2) Data Structures for internal representation of the Prolog database of facts and rules.
- 3) Parser and the associated syntax directed translation scheme for translation of clauses into the internal database.
- 4) Unification.
- 5) Control mechanism for procedure call and backtracking.
- 6) The cut operation.
- 7) Various predefined predicates.

We devote a separate chapter for the discussion on the implementation of each of the above.

The first is a simple task of converting an input stream of characters into tokens which act as terminal symbols for the parser.

Record formats are designed for each type of term in Prolog. This means that we have separate records for atoms, integers, functors and clauses. The facts and rules are represented internally by a tree of such records.

The dynamic declaration of operators significantly complicates the grammar and the design of the parser. The parser handles these operators through an ad hoc mechanism, akin to the infix to postfix conversion scheme. The syntax directed translation scheme basically builds the parse tree in terms of the database records.

Unification tries to match a goal with a clause head to create a common instance of the two. New terms are also built up as variables are bound to values. The representation of these new terms is through the structure sharing technique described later. Thus unification takes the place of tests and assignments in a conventional language.

Overview of Prolog Implementation

The control mechanism is based on the lush algorithm designed by Kowalski. As procedures are called, the current state of the system is saved on a set of stacks - the variable, trail and control stacks. However, because of the non-determinate nature of Prolog procedures, the state can't be popped off the stacks at the end of execution. Backtracking may cause a return to the procedure to try other solutions. Only when the procedure has no more solutions to offer can the state be removed. Thus backtracking strongly influences the design of lush, since one must be able to rapidly save and restore an earlier state of computation.

The cut operation makes determinate the execution of all procedures from the parent goal onwards. The control information required for backtrack to these procedures can therefore be discarded.

The predefined predicates are written as C functions. These predicates are also allowed to affect bindings of variables, and to have both determinate or non-determinate execution.

CHAPTER FOUR

THE LEXICAL ANALYSER

Introduction

Whenever the parser desires the next input token, it calls the lexical analyser. The lexical analyser scans the input line and recognizes tokens, which it passes upwards to the parser.

4.1 The Token Buffer

The interfacing between the parser and the lexical analyser is by means of the token buffer. This always contains the current token. When the next token is desired, the lexical analyser fills up the buffer area with the next token, which is thus made available to the parser.

The structure of the token buffer is as follows -

```
struct token
begin
  int start ;
  char type ;
  union
  begin
    int value ;
    char * strptr ;
  end
  data ;
end
```

The start field identifies the position on the screen where the current token starts. It is used for error handling - in case of an error in the current term, a mark is placed at the start position indicating where the error arose.

The type field identifies the token type. We have adopted the convention that in addition to the data types, punctuation marks are also declared in the type field. The data types possible are atom, integer, variable and string. Punctuation marks such as left parentheses, comma etc. are also declared in the type field.

The union field contains either the integer value, in case of an integer type, or a pointer to a character area that has a copy of the input string in case of an atom, variable or string type. In case of punctuation marks, this field is null.

Lexical Analyser

As an example the token for the atom 'ravi' is

```
begin
  type           -   TYP_ATOM
  start position -   10 (say)
  pointer to the string "john"
end
```

4.2 RULES FOR TOKEN RECOGNITION

Integers

They start with a digit and consist only of a sequence of digits. e.g. 20 .

Atoms

- are enclosed within single quotes e.g. 'anything'
- start with an alphanumeric and contain only alphanumerics. e.g. anything
- start with any symbol not a punctuation mark and contain non alphanumerics only. e.g. =..

Variables

They begin with an '_' or capital letter, and contain alphanumerics or underscores.

Strings

They are enclosed within double quotes.

4.3 PROGRAM OUTLINE FOR THE LEXICAL ANALYSER

skip over spaces, comments and lines until a non_white_space character is found.

switch (character)

begin

- | | | |
|-----------------------|---|--|
| case punctuation mark | : | set the token's type field to the appropriate value. |
| case double quotes | : | collect a string. |
| case digit | : | collect an integer. |
| case capital letter | : | collect a variable. |
| case underscore | : | collect a variable. |
| case small letter | : | collect an atom. |
| case single quotes | : | collect an atom. |

Lexical Analyser

```
case symbol          : collect an atom.
```

```
end
```

To read a new line, we check the current source of line input, which may be the terminal or some consulted file. A single line of characters is then retrieved from this particular source.

To store a copy of the input string in case of atoms, variables and strings, we use the function `get_char_space()`, which returns a pointer to a suitable memory area.

CHAPTER FIVETHE PROLOG DATABASEIntroduction

The set of clauses input by a Prolog programmer is converted into a data structure by the Prolog system and is stored in this form. This data structure, essentially a database of records, is used by the interpreter to answer queries.

At the core of the database are the record types that constitute it. These are as follows -

- 1 - Atom record
- 2 - Integer record
- 3 - Variable record
- 4 - Functor record
- 5 - Clause record
- 6 - Termlist record
- 7 - Goaltree record
- 8 - Goallist record

We shall consider each one in turn. For each record we describe

- 1- its structure
- 2- how it is created and used

5.1 ATOM RECORD

This is defined as
struct

```
begin
    char type ;
    char * name ;
    char is_predefined ;
    (* procptr)();
end
```

Type	Name	IS-predefined	Procptr
------	------	---------------	---------

Type

This is a single character field that specifies the type of the record, in this case, it is TYP_ATOM, a predefined constant.

Name

The name is a pointer to the character string for the atom name. When an atom is recognised by the lexical analyser, it allocates character space and places the name in that area. This area is pointed to by the atom record created by the parser. As in standard C convention, the name is null delimited.

Procptr

An atom name may be the name of a predicate. If this is so, then the procptr field is a pointer to the chain of clauses that defines the predicate. This field points to a C function in case the predicate is predefined.

The type of the field is 'pointer to function'. This is a pointer to a predefined C function. In case the field is to point to a clause it must be appropriately cast.

Is_predefined

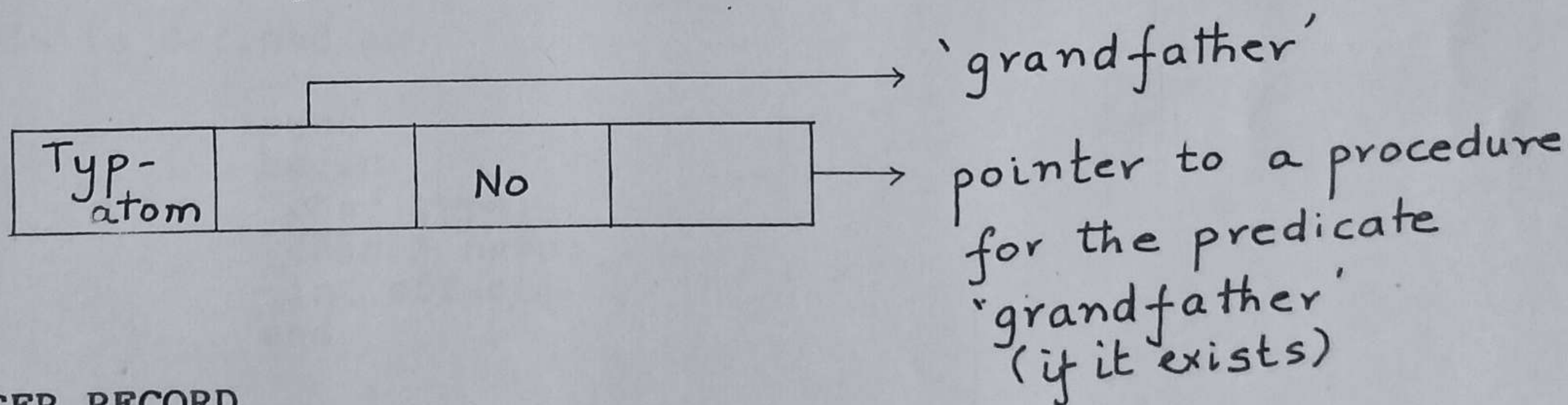
This is a flag that is set if the procptr points to a C function rather than to a clause list.

Creation

An atom record is created by a call to the function `get_atom_record(name)`, where `name` is a pointer to the atom's name, and the function returns a pointer to the atom record it allocates.

Example

The atom 'grandfather' is stored as

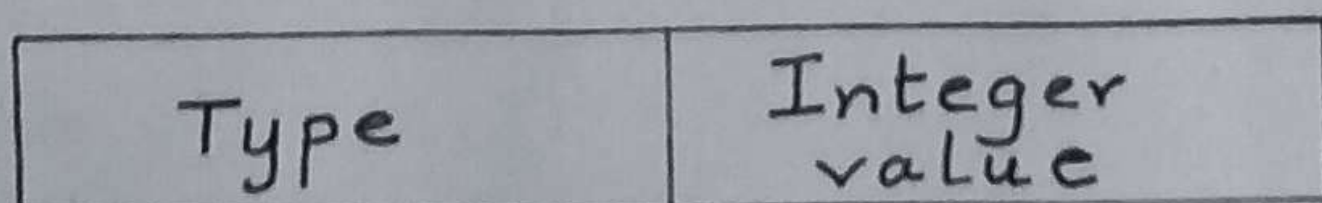


5.2 INTEGER RECORD

This is defined as

```

struct
begin
    char type;
    int integer_value;
end
    
```



Type

This is TYP_INTEGER.

Integer value

The value of the integer. Note that the field is of type 'int', so the largest number one can use in Prolog is $2^{32} - 1$.

Creation

This is by a call to `get_integer_record(value)`. Value is the integer value, and the function returns a pointer to the integer record it allocates.

Note

By Prolog convention, characters that are read in by the system using 'get' and 'get0' are also stored as integers. However, the integer value in this case is the ASCII value of the character read in.

Example

20 is stored as -

Typ- integer	20
--------------	----

5.3 VARIABLE RECORD

This is defined as

```

struct
begin
  char type;
  char * name;
  int offset;
end

```

Type	Name	Offset
------	------	--------

Type

This is TYP_VARIABLE.

Name

As in an atom record, this is a pointer to a null

Database Structure

delimited character string for the name of the variable. The name is used for debugging purposes only.

Offset

This is an integer which records the offset of the variable from the start of the stack frame for the clause in which the variable lies.

When the control reaches a clause, and the head is matched, a frame of size 'number of variables in the clause' is created on the variable stack. All variable instantiations are done in this frame on the stack. To find the currently instantiated value of a variable, it is hence necessary to determine the position of the variable in the stack frame for the clause. This is determined by adding the base of the clause frame on stack to the offset available in the variable record. Thus, the offset field provides us the means to determine the current instantiation status of a variable.

Creation

This is by a call to `get_variable_record(name)`. Name is the name of the variable and the function returns a pointer to the variable record allocated by it.

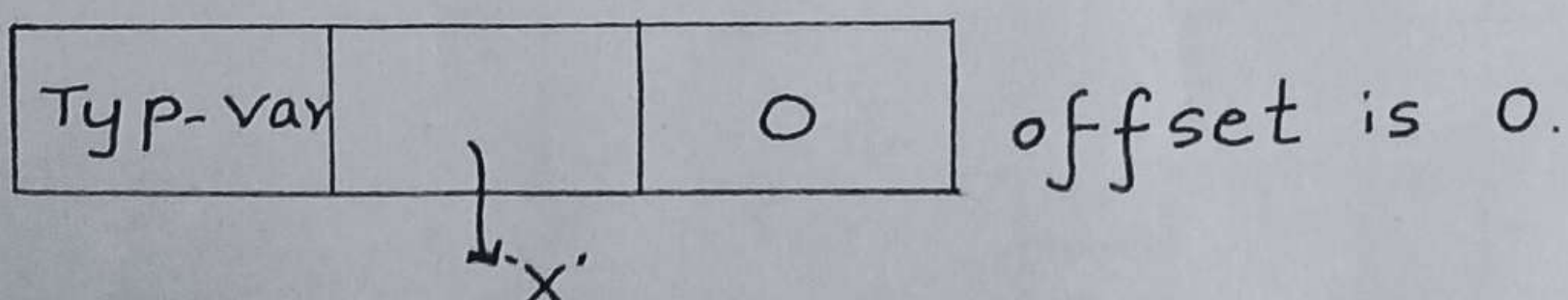
In Prolog variables have a scope extending over the clause in which they occur. Thus two or more occurrences of the same variable name in a clause must be bound to each other, i.e. whenever one of the is instantiated, so is the other. To ensure this, we have decided to let all variables of the same name in a clause to share the same clause record. This will automatically carry out the desired binding.

The function `get_variable_record` is intelligent enough to allow this sharing. For each clause, it maintains a variable table that contains the names of the variables in the clause. When a name is seen for the first time, it is entered in the table and a variable record is allocated for it. Subsequent occurrences of the name can be determined and they can easily be made to share the same variable record.

Example

x in clause 'grandfather'(x,y) :- father(x,A), father(

has the record.



5.4 FUNCTOR RECORD

This is defined as follows

```

struct
begin
    char type ;
    ATOM_RECORD * principal_term_ptr ;
    TERMLIST * argument_ptr ;
    int arity ;
end
    
```

Type	Principal term ptr	Argument ptr	Arity
------	--------------------	--------------	-------

Principal_term_ptr

A functor of the form $a(T_1, T_2, \dots)$ has 'a' as the principal term, an atom, and $T_1, T_2 \dots$ as a list of terms in the functor. The principal term pointer points to the atom record for 'a'.

Argument_ptr

This points to a list of arguments of the functor. The arguments are chained together by the termlist records, the first termlist in the chain is pointed to by the argument_ptr.

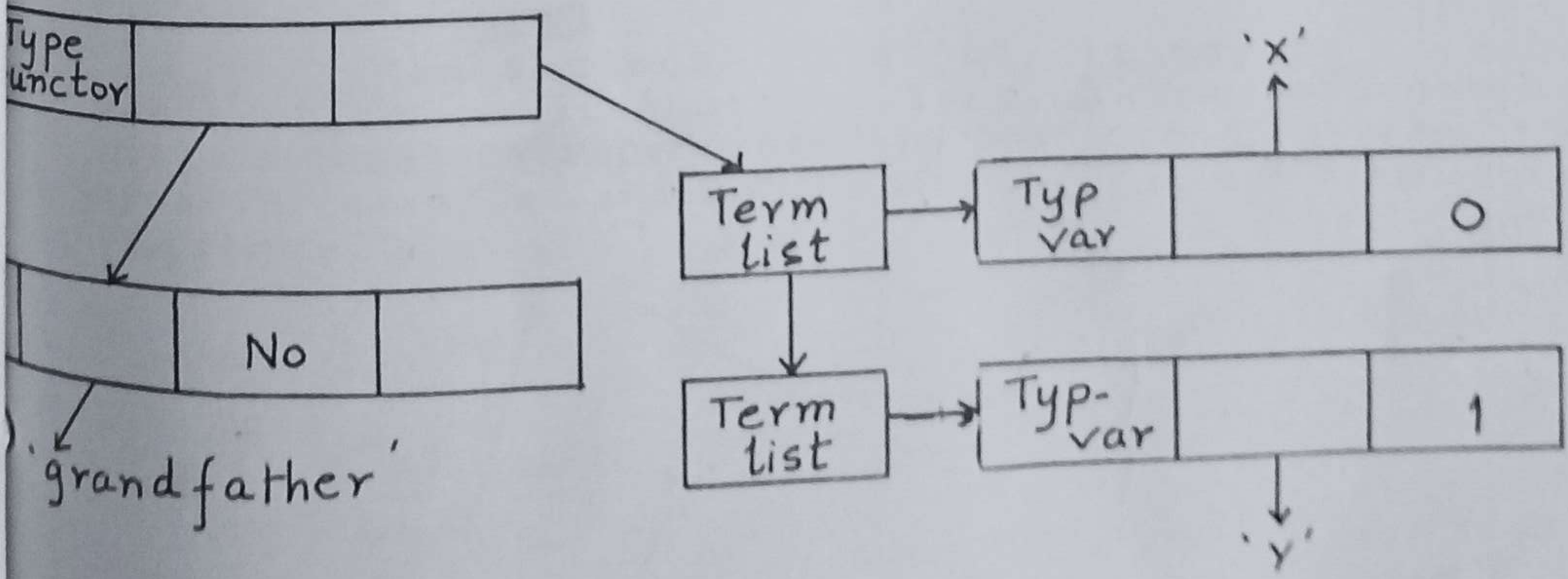
Creation

This is by a call to get_fn_record.

Example

grandfather(X,Y) in the clause
 grandfather(X<Y) :- father(X,A), father(A,X)

is saved as



5.5 CLAUSE RECORD

This has the structure

```

struct clause_rec
begin
char type;
FN_RECORD * head;
char * goal_ptr;
struct clause_rec * next_clause;
char arity;
end
    
```

Typ clause	Head	Goal_ptr	Next clause	Num-var
------------	------	----------	-------------	---------

Type

A clause record can be either of two types

- TYP_CLAUSE - if it is a clause for a predicate
- TYP_QUESTION - if it represents a query

Head

The head points to functor record. Consider a clause of the form

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

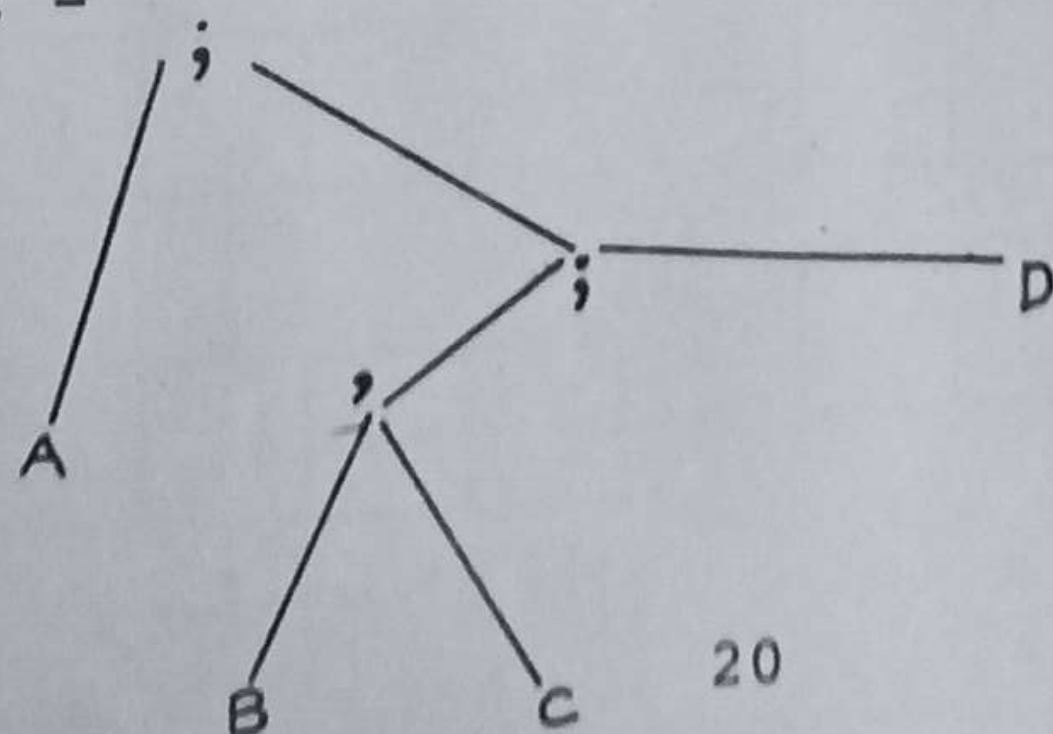
Here, the head is 'grandfather(X,Y)' and the goals are 'father(X,Z)' and 'father(Z,Y)'. We note that the head can be represented by a functor record. The clause record's head field points to a functor record that represents the head of the clause.

Each of the goals can also be similarly represented by a functor record.

Goal_ptr

The goal_ptr field points either to a goal tree or a goal list.

When a clause is input, it may be in arbitrary conjunctive or disjunctive form. These conjunctions and disjunctions are represented in the form of a tree, the goal tree. For example :-A;B,C;D where A, B, C, and D are goals, is represented as -



Database Structure

The datf parser parses these goals and constructs the goal tree. The goaltree is then linked up to the clause at the goal_ptr location.

The goal_tree is not suitable for the interpreter. Hence it is expanded into a disjunction of conjunctions form. Each of the disjunctions is then attached to a seperate copy of the clause record. Thus, the clause record after expansion will have only a conjunctive list of goals. This list is pointed to by the goal_ptr. For example the above can be represented as

A;(B,C);D

i.e. a disjunction of three conjunctive terms A,(B,C) and D. Thus three clause records are created during expansion, one with goal list A, the second with (B,C) and the third with D.

next_clause

The next_clause field points to the next clause in the list of clauses for a predicate.

num_var

When the control reaches a clause record, it has to create a stack frame for the clause, as explained earlier. The size of the stack frame is the number of variables in the clause. This is known from the num_var field of the clause record.

Creation

The clause record is allocated by a call to the function get_clause_record().

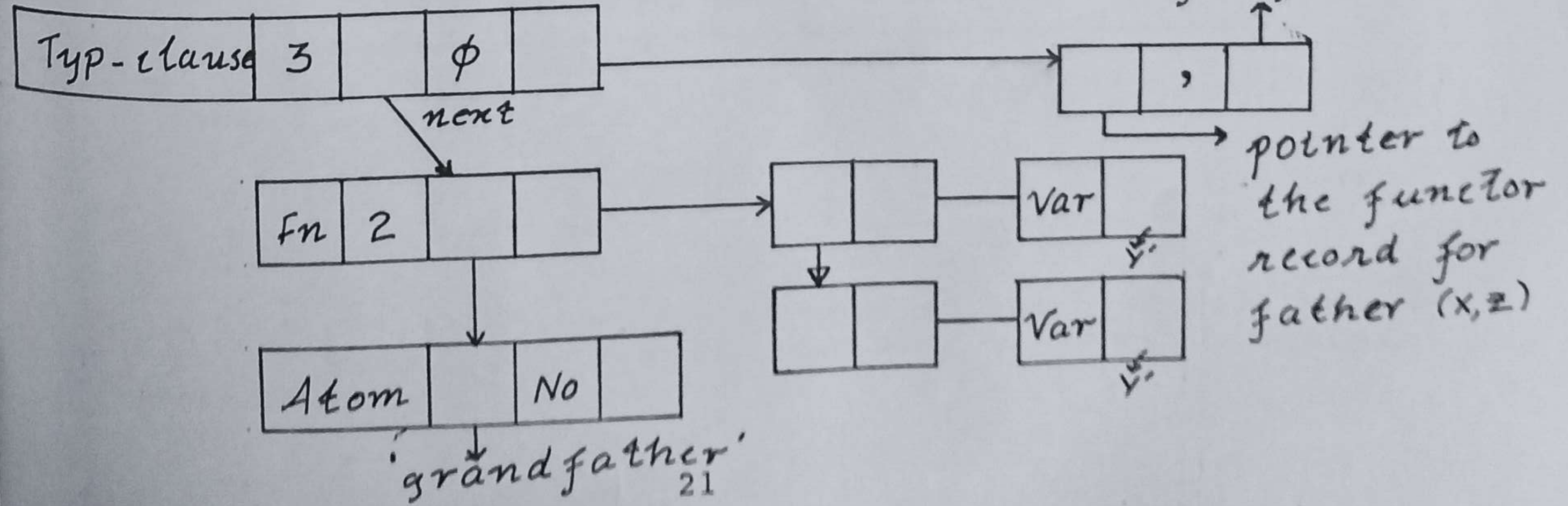
Example

For the clause

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

the clause record that will be formed is -

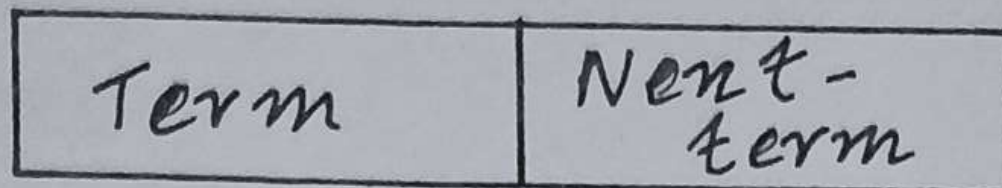
pointer to the functor record for father (Z,Y)



5.6 TERMLIST RECORD

This is used to link up terms in the list of terms that describe a functor. The record is

```
struct termlist_rec
begin
  char * term;
  struct termlist_rec * next_term;
end
```



Term

This points to the term of the functor, which can be an integer, atom or another functor.

Next_term

The next term in the termlist.

Creation

By a call to `get_termlist_record()`.

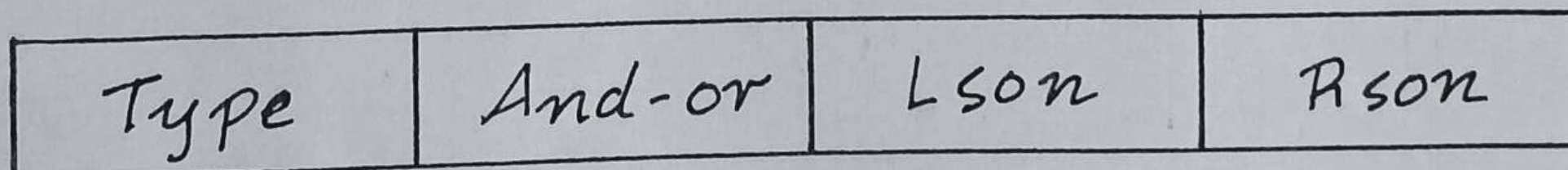
Example

Please look up the example for the functor record.

5.7 GOALTREE RECORD

The structure is

```
struct goaltree_rec
begin
  char type ;
  char and_or ;
  char * lson, *rson ;
end
```



Type

Type is `TYP_GOALTREE`.

And_or

It defines the type of node in the goaltree. This is set to `OR_NODE` or `AND_NODE` for the ';' and ',' operator respectively.

lson, rson

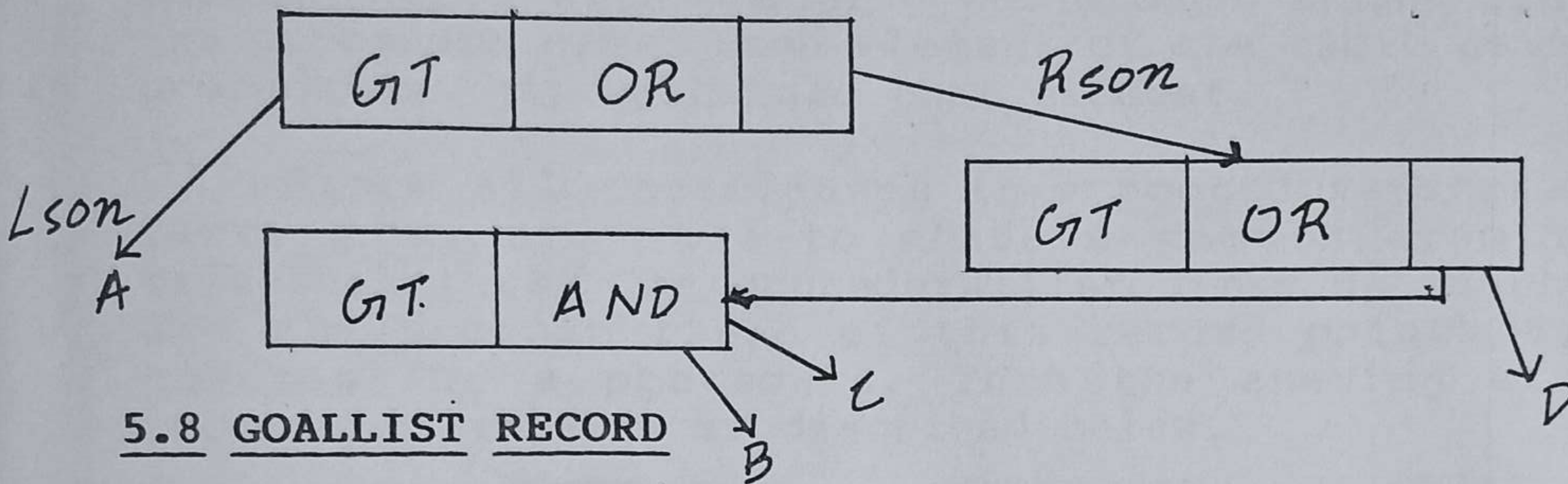
These point to the left subtree and right subtree of the node. If the left son is a goal, then the field points to a functor record, else it points to a goaltree record.

Creation

This is by a call to `get_goaltree_record()`.

Example

A;B,C;D is represented by the tree



5.8 GOALLIST RECORD

This has the structure

```

struct goallist_rec
begin
    FN_RECORD * goal;
    struct goallist_rec * next_goal;
end
    
```

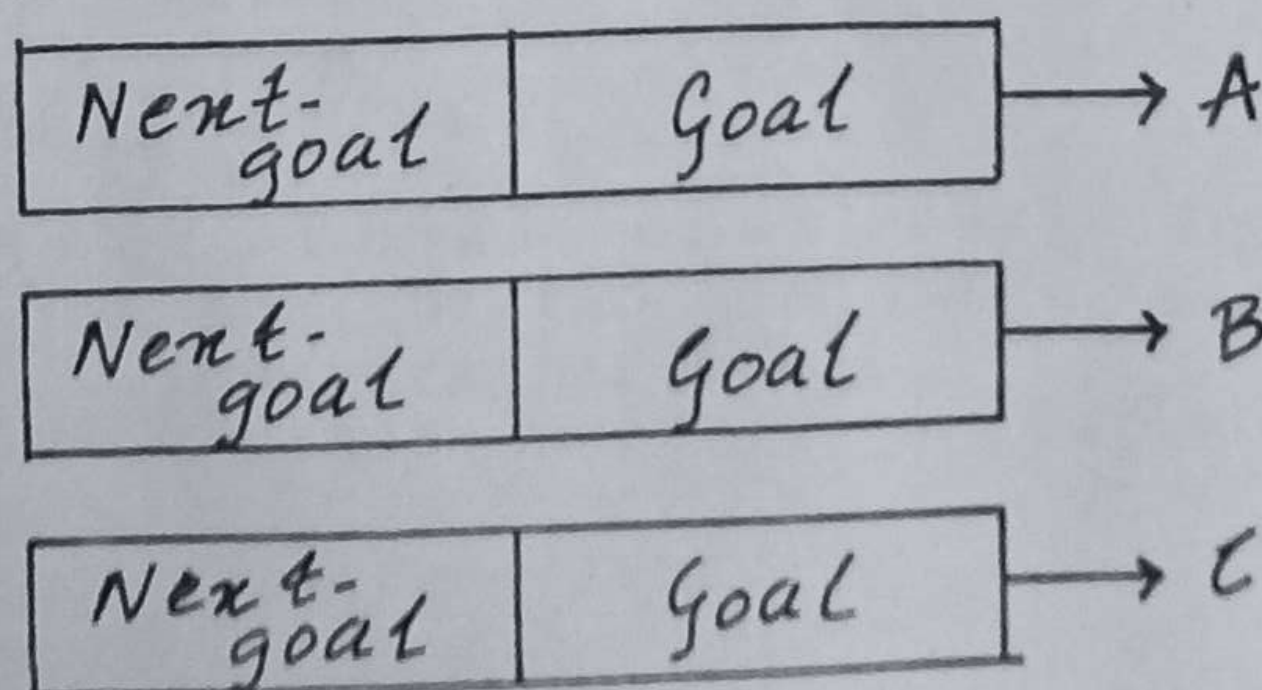
Goal points to the functor record for the current goal.

Next_goal is a pointer to the next goal in the list.

Creation is by a call to `get_goallist_record()`.

Example

The list of conjunctive goals A,B,C is represented as

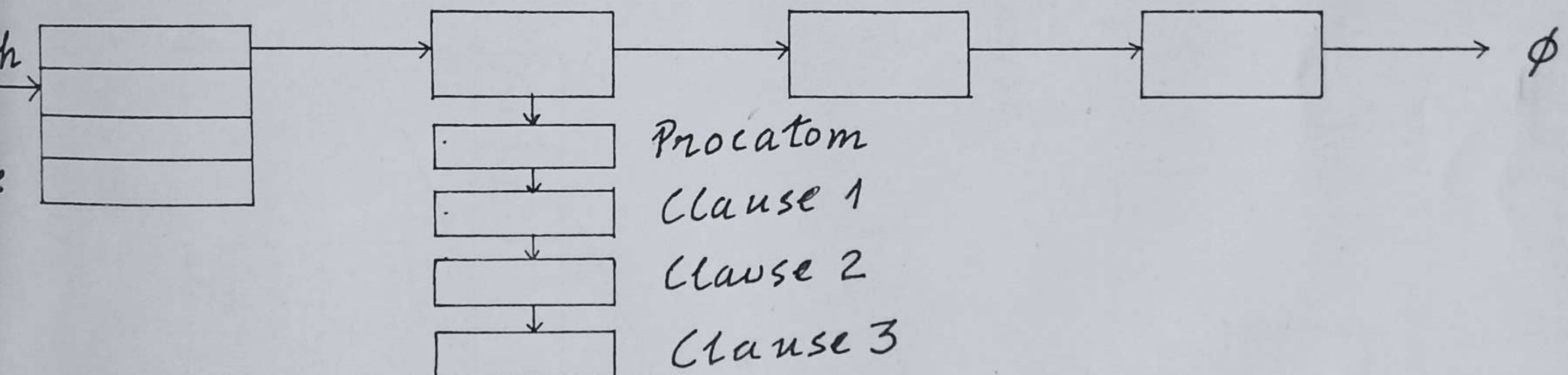


5.9 THE PROCEDURE TABLE

All the clauses in the database are grouped into procedures according to the name of the principal term of the head. A procedure is hence a list of clauses that share a common name. When a goal with this name is to be satisfied, a call to the procedure is made. Thus control passes from the point of call (a goal) to the first clause in the procedure called. To enable a rapid transfer of control, a link is maintained from the point of call to the procedure for the call. This is via the `procptr` field of the atom record for the principal term of the goal. The link is created while the clause is being converted into the database format.

Procedures are stored in the procedure table. This is a hashed table, with hashing being done on the first character of the procedure name. Each element of the table points to a list of procedures that hash onto that element.

Since all the clauses in a procedure share the principal term, they are made to share a common atom record for the principal term. The procedure list links up to this atom record, and the `procptr` field of this record points to the chain of clauses for a procedure. Thus the ensuing structure of the procedure table is as described below.



```

A proc entry record is
struct procedure
begin
  ATOM_RECORD * proc_atom;
  struct procedure * next_proc;
end
  
```

The `procatom` field points to the common(shared) atom for the procedure.

5.10 EXAMPLE

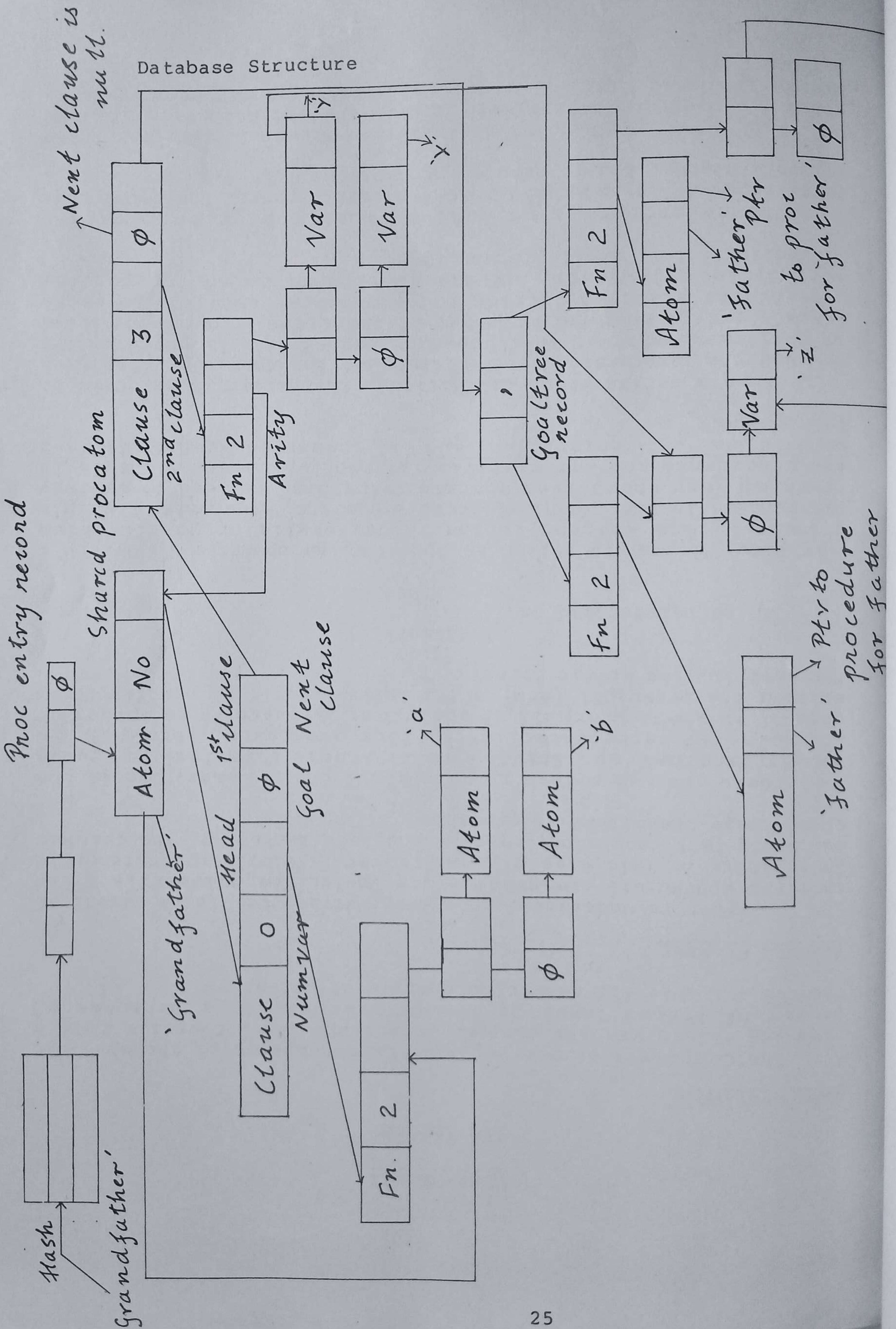
As a final example, the complete database representation for the two clauses

```

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

is given overleaf -

Database Structure



CHAPTER SIXTHE PARSERIntroduction

The component of our implementation that took the maximum time and effort was the parser. It effects the conversion from a stream of tokens input by the programmer, to a database of records suitable for answering queries efficiently.

6.1 The Grammar

In order to understand the parser structure, it is essential to first appreciate the grammar for Prolog. The grammar is given in Appendix 1. This form of the grammar is easily understandable but needs left recursion removal and left factoring. The grammar after these operations is presented in Appendix 1A.

There are two points to note about the grammar. One is the pseudo-terminal 'op'. Structures (or atomic formulae) in Prolog are normally represented in the form `atom(term1,term2,...)`. In case of an atomic formula of arity 1 or 2, there is an option of writing the principal term in the infix, prefix or postfix form. For example, the atomic formula

	<code>likes(a,b)</code>
can be represented as	<code>a likes b.</code>

Here, 'likes' is called an operator. An operator is merely a principal term written in the infix, prefix or postfix form rather than in the canonical form. Since the 'arguments' for this operator may themselves contain operators, we need to specify the precedence and associativity of each operator. All this is done in the 'op' declaration, and all operators must be defined before use.

Given a series of op definitions, the parser is expected to compute the 'expression tree' defined by the operators. The task is similar to that of an expression parser in a language such as Pascal, except that here the set of operators is dynamic. Thus, none of the standard parsing techniques can be used. To parse such dynamic grammars, we have developed an ad hoc parsing mechanism that we shall discuss later.

The second point to note is that the grammar can be partitioned quite naturally into 6 separate parts. The first part parses a clause, and it handles productions 1 to 4.

Another parser handles the DATF non terminal. ATFs and TERMS require a distinct parser each, since they both contain the 'op' pseudo terminal. Finally, lists and structures can be handled by a parser each.

We had initially wanted to have the entire parser in the form of a predictive parser. This was not possible because of the dynamic operator problem. Further, it became apparent that DATF could be parsed quite efficiently and naturally by an operator precedence parser. Hence, we retained predictive parsing for the clause, list and structure parsers. The DATF parser is operator precedence, and the ATF and TERM parsers are ad hoc mechanisms.

The clause, list and structure parsers share a global predictive parsing table and each accesses a set of predefined rows of the table. The parsing table for these parsers is given in Appendix 2. The design strategy of the table is standard, and we urge the reader to look up ppl84 of Reference 1 for further details. The lists of First and Follow required by the table are available in Appendix 3.

6.2 CLAUSE PARSER

The clause parser handles the following productions -

```
CLAUSE -> ATF C
CLAUSE -> ?- DATF.
```

```
C -> .
C -> :- DATF.
```

The parser is a predictive parser and uses the first two rows of the predictive parsing table. On seeing the First of a DATF or an ATF non-terminal, a call is made to the datf parser or atf parser respectively.

The semantic actions of building up the database are incorporated into all the parsers. When each production is detected, the associated semantic action is embedded into the code that handles the production. For example, when

```
CLAUSE -> ATF C
```

is detected, a new clause record is allocated and the atf parser is called. The atf parser parses the head of the clause, and this is then linked up in the clause record.

A significant problem is to develop a mechanism for passing of parameters between parsers. For example, when the clause parser calls the ATF parser, the clause parser must get a pointer to the section of the database created by the ATF parser. To handle parameter passing in a consistent fashion, we have used the concept of location. Whenever any parser is called, a pointer to a location is passed as a parameter. After the parser completes its action, the location is made a pointer to the section of the database created by that parser. Continuing with our example, when the atf parser is called, the location passed to it is the address of the 'head' field of the clause record newly created by the clause parser. The atf parser parses the

Parser

head of the clause and then links up the functor record it creates for the head onto the location. In this way the head field of the clause record is filled up. Using a similar strategy, the entire database of records is built up, with each parser contributing a section to it by linking up the section at the appropriate location.

Clause Parser Algorithm

The overall algorithm followed by the clause parser is as follows -

create a new clause record.

reset variable count and variable table for this clause.

push \$ and CLAUSE non terminals onto the clause parsing stack(CPS).

switch on the top element of the CPS.

begin

case \$: set the number of variables found so far in the num_var field of the clause.
enter the clause into its procedure.

default: examine the input character and switch on the appropriate entry in the predictive parsing table.

begin

case CLAUSE -> ATF C:
call atf parser and link up the head.
push C on stack.

case CLAUSE -> ?- DATF. :
set clause type to TYP_QUESTION.
call the datf parser and link up the list of goals parsed by it .

case C -> .
nothing to do.

case C -> :- DATF.
call datf parser and link up the list of goals parsed by it.

end

end

Note that at each stage, if the parser expects any terminal in the input, and if such a terminal is not actually found, then an error is signalled.

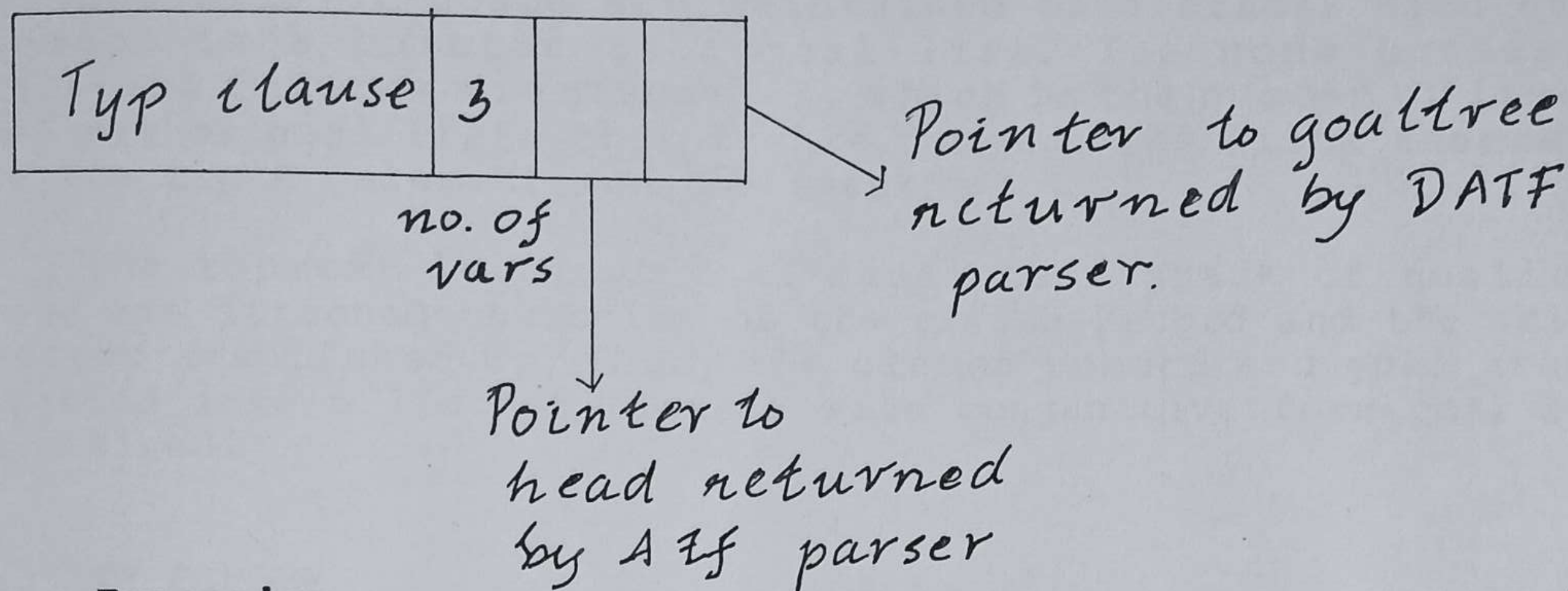
The action performed by 'enter clause into procedure' is to link up a clause into the procedure for the clause. The function

parser

checks up if the procedure for the clause already exists. If so the clause is linked up, else, a new procedure is created and the clause is placed in it.

As an example, we consider the database records formed when the following clause is parsed -

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

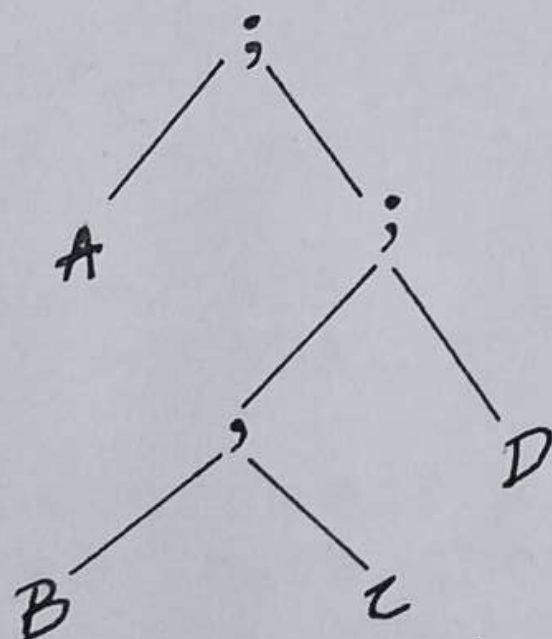


Expansion

The user may specify the goals in an arbitrary disjunct of conjuncts form. This form, when parsed, creates a goaltree structure. For example, the set of goals

A;B,C;D

creates the goal tree below-



Though the tree is suitable for representation of the programmer input, it makes the interpretation strategy difficult. This is because the interpreter is based on true Horn Clauses, which do not allow disjunctions. To allow for ease of programmer input as well as for efficient interpretation, we have resorted to a stratagem of 'expansion'. Given a goal tree, we convert it into a set of goal lists, such that the goal lists are of conjuncts only, and each goal list expresses one disjunct of the goal tree. For example, the above set of three disjuncts can be expressed as three goal lists i.e. A, (B,C), and D - which are all in conjunctive form.

parser

The method used to convert from the goal tree representation to the goal list representation is outlined below.

The algorithm essentially operates on each node of the goal tree. If the node is a leaf, then an element of the goal list is created and returned. For a ';' node, the union of the set of goal lists of the left and right sons is returned. For a ',' node, the cross product of the set of goals of the left and right sons is returned.

The sets of goals are maintained on a stack. Each stack element is a pointer to a goal list. The node processing algorithm returns an integer, k , which is the number of lists in the set of goal lists that it has formed. The lists themselves are the top k elements on the stack.

The topmost level of call receives a stack of goallists. These are attached to copies of the clause record and the clause records are linked up. Thus, the clause record and goal tree is expanded into a list of clauses with conjunctive form goal lists as desired.

6.3DATF PARSER

This handles the following productions -

DATF \rightarrow CATF D

D \rightarrow epsilon

D \rightarrow ; DATF

CATF \rightarrow GOAL CATF'

CATF' \rightarrow , CATF

CATF' \rightarrow epsilon

GOAL \rightarrow ATF

GOAL \rightarrow \$DATF†

Since the grammar is nearly identical to the grammar for expressions, we have chosen the standard parser for expressions i.e an operator precedence parser.

As in the clause parser, the semantic actions are embedded in the parser itself. Corresponding to the expression tree a goal tree of goals is built up.

Incoming operands are placed in the operand stack. Whenever an operator of priority lower than that of the operator on the top of the operator stack appears, then the stack is popped. Semantic actions are taken depending upon the type of operator popped off. This is best explained by the table below. The datf parser is merely the expansion of the table into C code.

		I N S T A C K			
O U T S I D E					
	,	COM-REC	SEMICOLON REC	PUSH ,	PUSH ,
	;	COM-REC	PUSH ;	PUSH ;	PUSH ;
	{	PUSH \$	PUSH \$	PUSH \$	PUSH \$
	{	COM-REC	SEMICOLON REC	ERROR	POP \$
.	COM-REC	SEMICOLON REC	FINISH	ERROR	
any other symbol		CALL ATF PARSER			

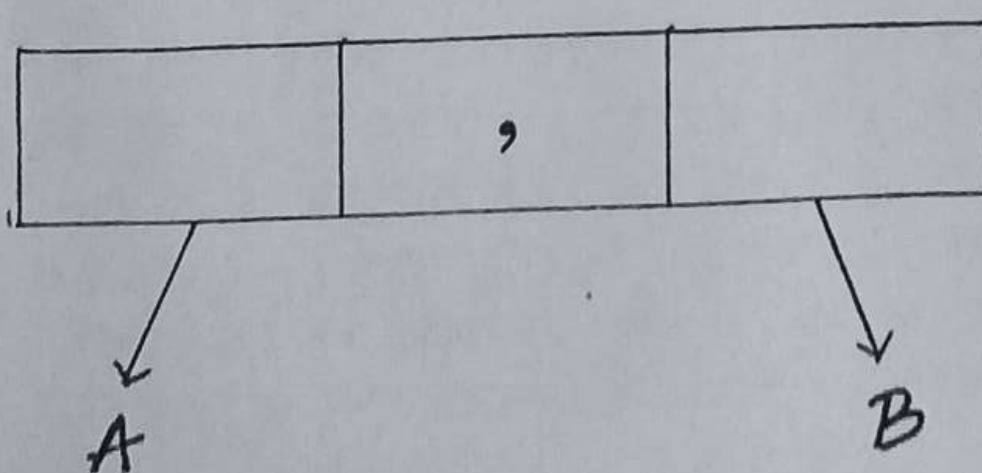
Note that ',' is left associative and ';' is right associative.

COM-REC - create a comma record, ie a goaltree node with ',' as the operator.

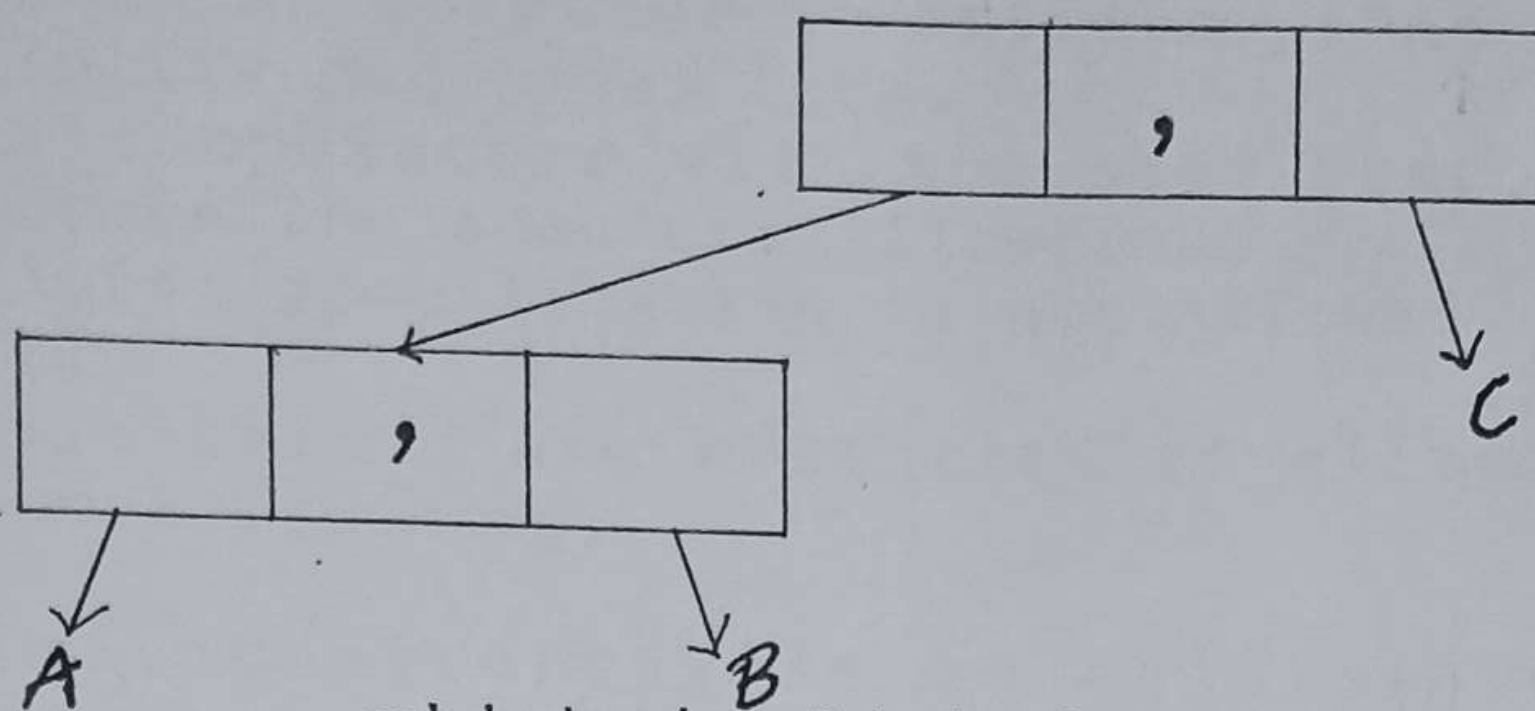
SEMICOLON REC - create a semicolon record in the goal tree.

To build a goaltree record, we get a goal tree node and set the appropriate type. Then, the topmost two operands in the operand stack are linked as the left son and the right son.

For example, for a set of goals A, B, C - when we reach the second ',', the operand stack has A and B with top of the operator stack being ','. The table tells us that the action to be taken is to create a comma record. This is done and A and B are linked up as its sons as shown-



Then, this is placed on the top of the operand stack. C is pushed on top of this. When '.' is seen, another comma record is created with (A,B) and C as its sons. This gives us the desired data representation -



which is linked to the goal field of the current clause.

6.4 ATF AND TERM PARSERS

These parsers handle the following productions -

- ATF -> STRUCTURE
- ATF -> LIST
- ATF -> TERM 'op' TERM
- ATF -> TERM 'op'
- ATF -> 'op' TERM

- TERM -> STRUCTURE
- TERM -> LIST TERM'
- TERM -> 'OP' TERM TERM'
- TERM -> integer TERM'
- TERM -> variable TERM'
- TERM -> atom TERM'
- TERM -> (TERM) TERM'

- TERM' -> 'op' TERM TERM'
- TERM' -> 'op' TERM'
- TERM' -> epsilon

These productions incorporate a TERM' nonterminal which is used to remove left recursion of the form

TERM -> TERM op TERM.

The problem with the above productions, as has already been explained, is that the associativity and precedence of the 'op' pseudo-terminal can be dynamically changed. Hence the expression tree formed by using the 'op' term depends upon the current status of the op declarations. We came to the conclusion that an ordinary parsing strategy could not handle this problem. Hence, we

decided to implement an ad hoc parsing mechanism for these productions.

We place three restrictions -

- once an operator is defined, then it cannot be used in its canonical form.
- all operators with the same precedence level must share the same specification.
- 'yfy' specification is not allowed.

These conditions are essential to allow us to parse the expressions unambiguously.

The parsing strategy is to maintain two stacks - an operator stack and an operand stack.

- Variables, structures and atoms not declared as operators are pushed onto the operand stack.
- Those atoms which are defined as operators are pushed onto the operator stack if they have higher precedence than the top element of the operator stack. If not, a tree node with the top element of the operator stack as its operator and with the top 1 or 2 elements of the operand stack as its son(s) (depending on the arity of the operator) is created. /* note that in Prolog, an operator with higher precedence has lower priority */

The specification of the operator is then checked. If it has an 'x' type on the left or right of the operator, then that subtree is checked for operators of equal precedence. For example, for a specification of xfy, the left subtree is checked for an operator of the same precedence.

The outline of the algorithm for the atf and term parsers is given below.

```
t = current token.
switch on t.
begin
  case & : /* note that & is the cons symbol for list
            representation */
            call the list parser

  case of an element in the follow of ATF or TERM :
            go to return code.

  case string :
  case [ :
            call the list parser.

  case ( :
            push it on the operator stack.
```


case) : /*) is not only in the production for TERM,
it is also in the follow of TERM. In any
case .. */

pop the operator stack until the operator on top
is either (or \$.

If the stack has \$, then the) must have been in
the follow of term, so we leave) as the current
token to be handled by the upper layer and goto
return code.

If (is found on stack then it is popped, the
input token is consumed, and the next token is
called for.

case atom :

If it is an operator - it is pushed on stack
depending upon its precedence, as explained above.

If it the first term in a structure, the structure
parser is called.

If it is a simple atom, it is pushed on the
operand stack.

case integer :

case variable :

It is pushed onto the operand stack.

end

return code :

pop the operator stack creating tree nodes as
necessary until the \$ operator is reached.
return.

Note that in order to determine that an atom is indeed the
first term of a structure, it is necessary for us to see whether
the token immediately following the atom is a '('. This requires
a look ahead of one. To provide this lookahead, we do buffering of
the token stream. At any given time we keep in the buffer the
current token as well as the next token that was read in. To do a
lookahead, all we do is to look at the next token which is
already present in the buffer.

As an example of the above parsing strategy, consider the
expression

$$a(b,c) + d \% e$$

where we have already defined + to have
precedence 100 and specification xfy, and % to have precedence
150 and specification yfx. The sequence of stack states is shown
below.

	+	+	%	%	
a(b,c)	a(b,c)	d a(b,c)	a(b,c) +d	e a(b,c)+ d	
			$\begin{array}{c} + \\ / \quad \backslash \\ a(b,c) \quad d \end{array}$	$\begin{array}{c} + \\ / \quad \backslash \\ a(b,c) \quad d \end{array}$	$\begin{array}{c} \% \\ / \quad \backslash \\ e \quad + \\ \quad / \quad \backslash \\ \quad a(b,c) \quad d \end{array}$

Note that the code for the term and atf parsers is identical. Why is this so? The major portion of the code in the term parser deals with the parsing of structures in the expression form. Since atfs are syntactically identical to structures, the same code is also needed by the atf parser. Thus the two parsers are identical.

The major difference between the two parsers is that whereas an atf parser must return a functor record, a term parser may return a variable, atom or functor value.

Also note that the algorithm above resembles the operator precedence parsing algorithm. The difference lies in the fact that in the operator precedence parser, the operators and operands can be easily differentiated, whereas in the existing situation, this is not trivially possible. Further, in an operator precedence grammar, we can statically create the operator precedence table, whereas here, one must maintain a dynamically varying precedence table.

6.5 STRUCTURE PARSER

The structure parser handles the following productions -

STRUCTURE -> atom (STERMLIST)

STERMLIST -> TERM S

STERMLIST -> epsilon

S -> , STERMLIST

S -> epsilon

These productions actually have some redundancy .The above productions are equivalent to

STERMLIST -> TERM , STERMLIST

STERMLIST -> epsilon

We had initially planned to disallow structures with null termlists. Hence, S was introduced to ensure that the termlist could never be null. However, we later found they we needed a null termlist in case of calls to predefined predicates with no parameters. Hence, STERMLIST -> null was introduced, leading to a redundancy. Since removal of the redundancy would require a major modification in our code, we have avoided this change. In any case, the grammar is still correct.

The structure parser is a predictive parser and it uses rows 3 and 4 of the parser table. The standard predictive parsing scheme is adhered to. The outline of the algorithm for the parser is as follows -

initialize stack to

atom
(
STERMLIST
)
\$

create a functor record for the structure.

switch on top of stack.

begin

case \$: return.

case (:

case) :

case , : remove the corresponding token from the input stream .

case atom :

create an atom record and place it as the principal term of the functor.


```

default :
  look at the current input token and switch on the
  predictive parsing table.
  begin

    case STERMLIST -> TERM S :
      parse the term and link it to the current
      termlist record.

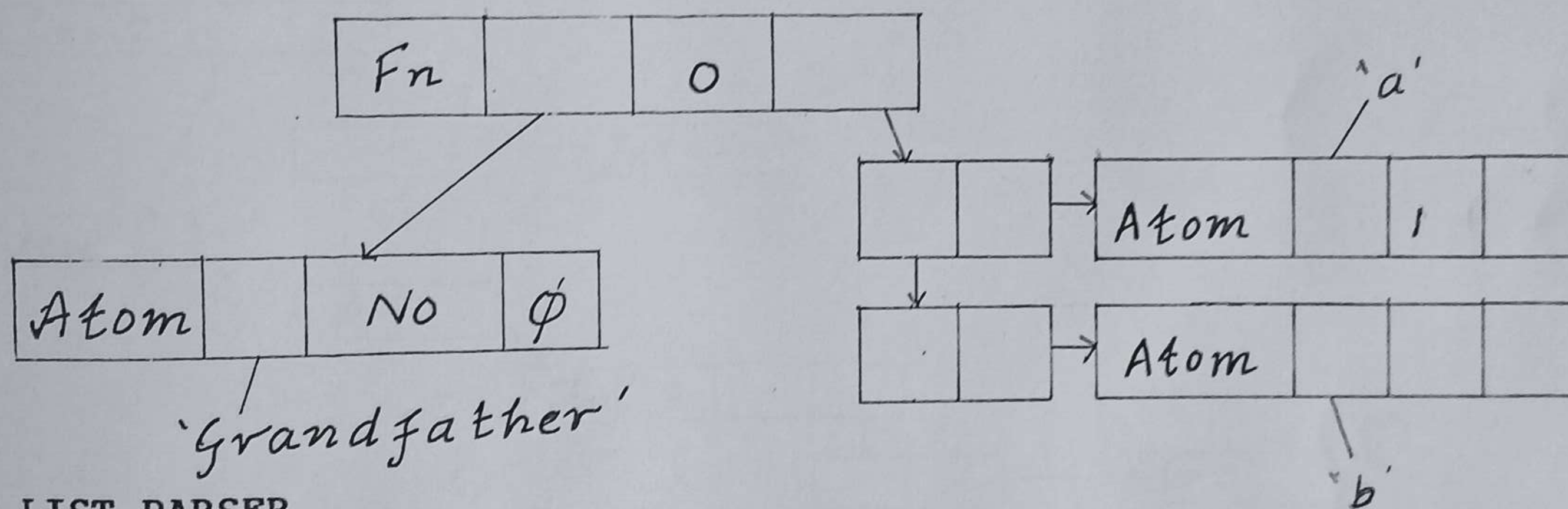
    case STERMLIST -> epsilon :
      end the termlist chain.

    case S -> , STERMLIST :
      get a new termlist record and link it up to
      the existing termlist chain.

    case S -> epsilon :
      terminate the termlist chain.
  end
end

```

As an example, the data structure formed for the input structure 'grandfather(a,b)' will be -



6.6 LIST PARSER

The list parser handles the following productions -

```

LIST -> string
LIST -> & ( TERM , LIST )
LIST -> [ L

L -> ]
L -> TERM M

M -> ¶ N
M -> , LTERMLIST
M -> ]

N -> variable
N -> LIST

LTERMLIST -> TERM LT

```


parser

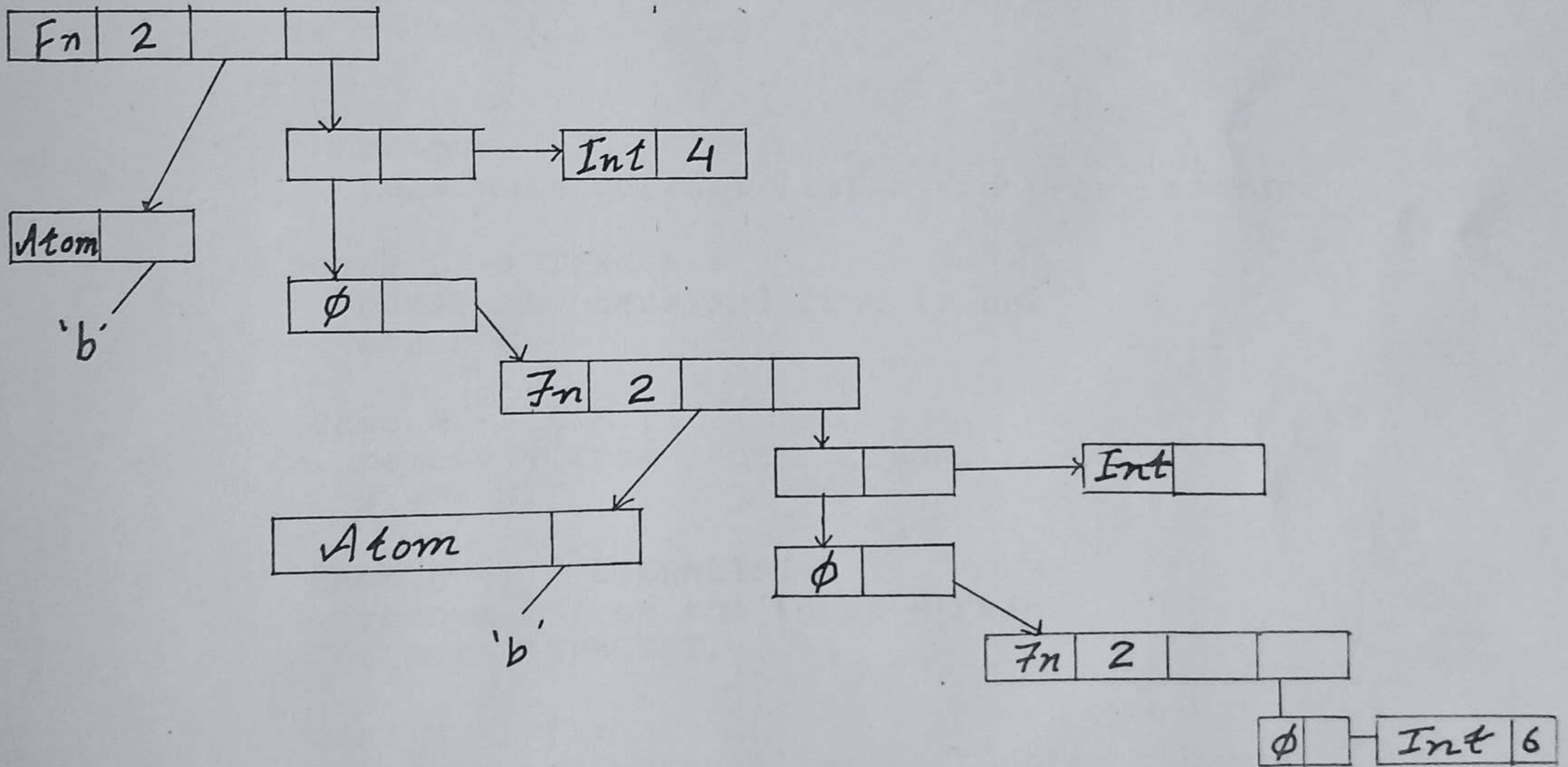
LT -> , LTERMLIST
 LT -> epsilon

The list parser is a predictive parser and follows the standard predictive parsing algorithm. It uses rows 5 to 10 of the predictive parsing table.

The parser has to deal with lists that can be input in a variety of forms. These are

- 1- & (TERM , LIST) e.g &(4, &(5, &(6, [])))
- 2- [TERM, TERM ...] e.g [4, 5, 6]
- 3- [TERM ¶ LIST] e.g [4¶[5¶[6¶[]]]]
- 4- string e.g "456"
- 5- [TERM ¶ variable] e.g [4 ¶ X]

Though externally different, they all form a chain of functor records with name & and arity 2. For example, all of the above are represented as -



Note "456" is an exception to the above. It will be stored with 4, 5 and 6 as integers but in their ASCII form.

The outline of the algorithm used for parsing is given below -

initialize the stack to

LIST
\$

switch on the top of stack.

begin

case \$: return.

case) :

case] : remove this token from the input stream and get the next token.

default :

switch on the predictive parsing table.
begin

case LIST -> string :

make a list of functor records as described above, with each character in the string being stored in ASCII form.

case LIST -> & (TERM , LIST) :

remove & from the input stream.
parse the term by calling the term parser.
link up the term.
remove (from the input stream.
stack) and LIST.

case LIST -> [L:

remove [and stack L.

case L ->]

remove].
terminate current list of functor records.

case L -> TERM M :

parse the term and link it up.
stack M.

case M -> ¶ N :

remove ¶ from input stream.
stack N.

case M -> , LTERMLIST :

remove , from the input stream.
stack LTERMLIST,

case M ->] :

terminate current list of functor records.

case N -> variable :

link up the variable in the current list of variable records.

case N -> LIST :

stack LIST.

case LTERMLIST -> TERM LT :

parse and link up the term.

Parser

```
stack LT.
```

```
case LT -> epsilon :  
    terminate current list of functor records.  
end
```

```
end
```

The list parser maintains a global called `list_posn` which always points to the location on the current list of functor records where the next functor record is to be added.

This concludes our discussion of the various parsers. The discussion should explain exactly how the database is created. We now examine how the database is used for unification and interpretation.

CHAPTER SEVEN

UNIFICATION

We first discuss the structure sharing technique used to represent values of variables build up during unification. This is followed by an explanation of the unification algorithm itself.

7.1 STRUCTURE SHARING

The key problem solved by structure sharing is how to represent an instance of a term occurring in the original source program. This instance is built up during unification. We call the original term a source term and the new instance a constructed term. Structure sharing represents the constructed term by a pair :

(source term, frame)

where the frame is a vector of constructed terms representing the values of the variables in the source term. The relative position of a variable's value within the frame is given by its offset number. A variable that is unbound is given the special value undefined. All bound variables have the value of the term they are bound to.

For example, given the source terms

```
tree( a, X, Y ).
tree( A, B, C ).
```

the constructed term

```
tree( a, tree( A, b, C ), [] ).
```

is shown in figure 7.1(a).

If the source term is a constant, or undefined, then the frame pointer becomes meaningless, since there are no variables to be referenced.

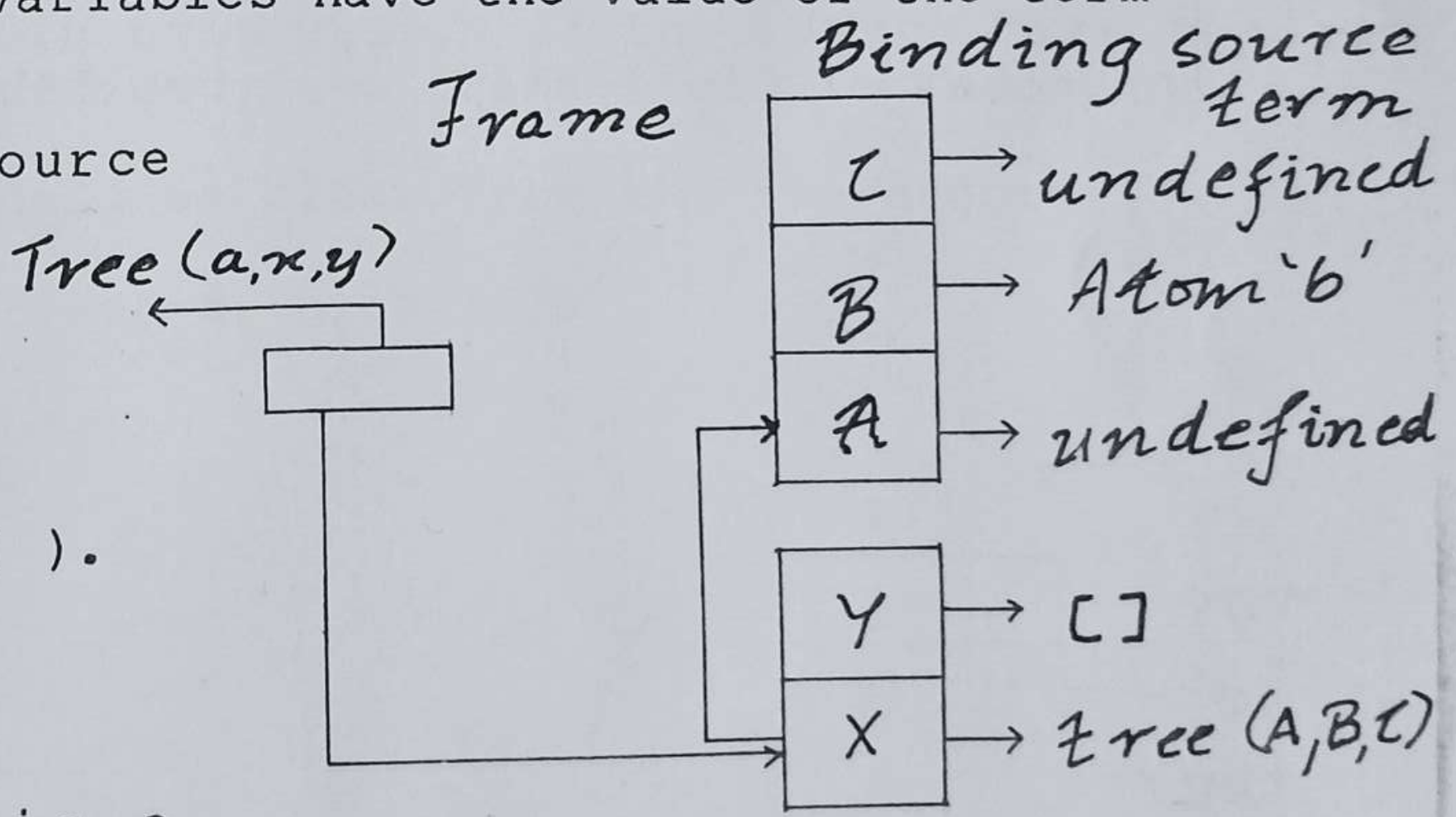


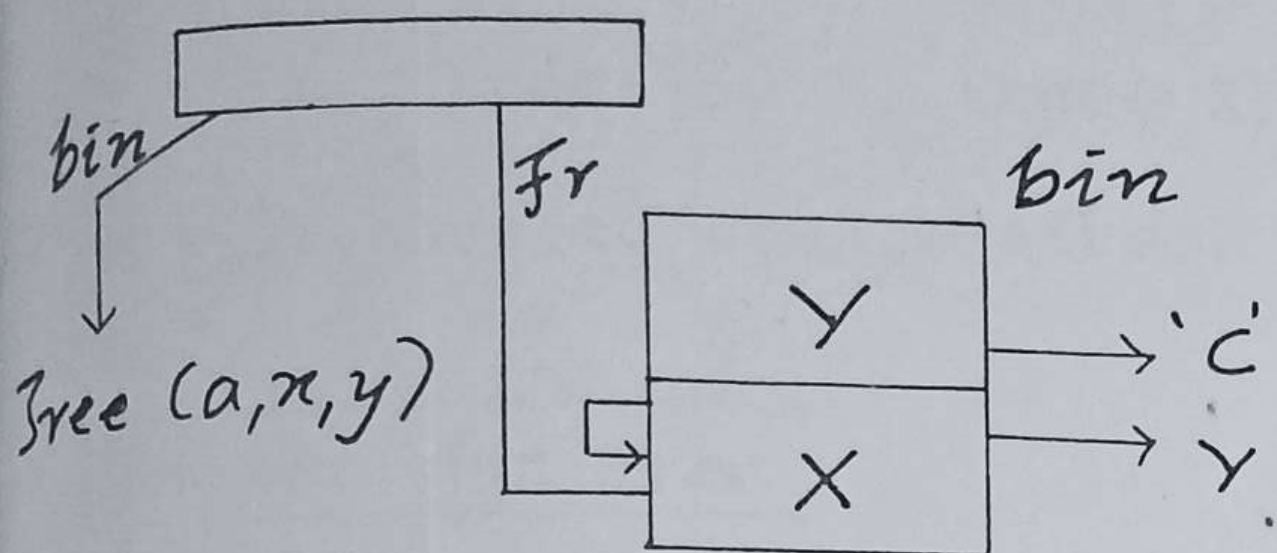
Figure 7.1(a)

Unification

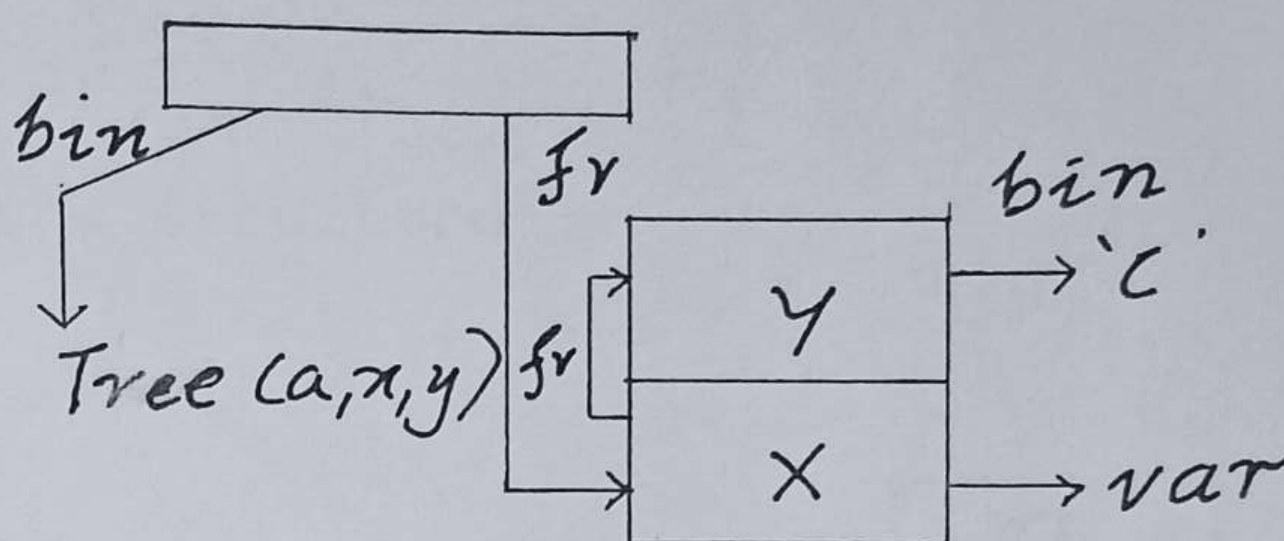
If the source term is a variable, a simplification is possible. The frame pointer can directly point to the cell containing term need not be stored. Consider the constructed term

$tree(a, X, Y)$.

X being bound to Y, and Y to the atom 'c'. Two representations are possible :



7.1(b)



7.1(c)

In 7.1(c) the constructed term for X consists simply of a pointer to the cell of the variable to which it is bound. The source term only contains an indication that X is bound to a variable.

Thus both the fields of a constructed term are important only if the source term is a skeleton. In such a case the constructed term is called a molecule. Only the frame field is important if the source term is a variable. The constructed term is called a reference in this case. If the source term happens to be a constant or undefined, the frame field is redundant.

These concepts should be clear from the representation of

$tree(a, X, tree(X, b, X))$.

shown below :

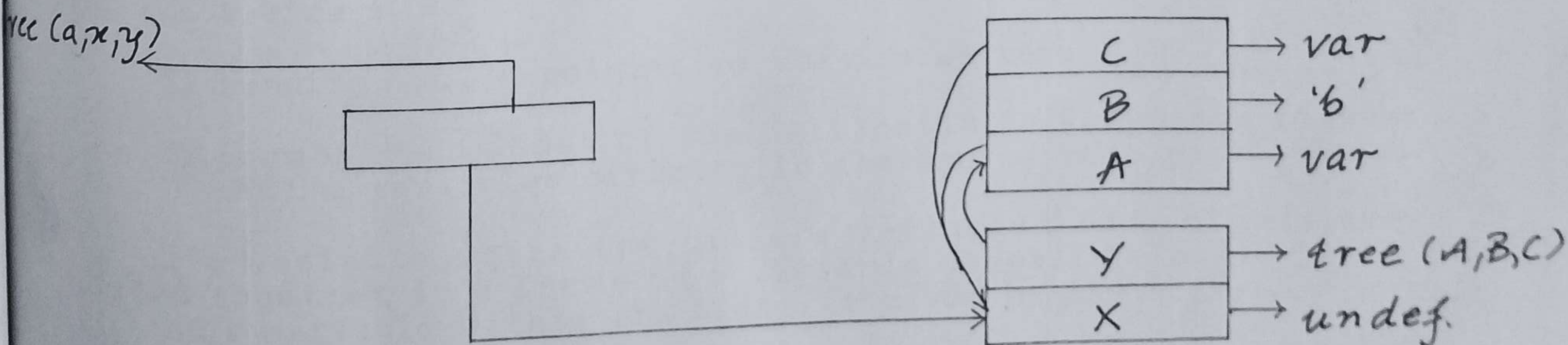


Figure 7.1(d)

Unification

The advantage of structure sharing representation is that the cost (in terms of both storage and time) of constructing new terms from skeletons occurring in a clause, and therefore already existing in the database, is, at worst, proportional to the number of distinct variables in those skeletons. New storage is required only for the variable cells and no copies of terms have to be created. In contrast if we were to represent the constructed terms using the same data structures as for source terms, the cost will be proportional to the number of terms in the skeleton, since a copy of the original terms will have to be created. For example, to represent the constructed term

tree(a, X, tree(X, b, X)).

we will have to create afresh the structure :

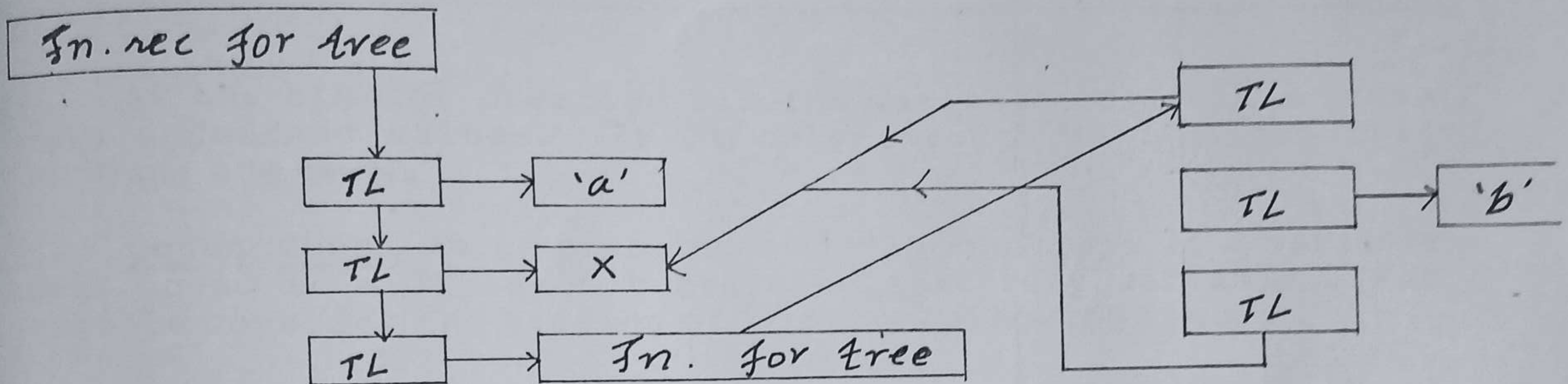


Figure 7.1(e)

The only disadvantage of structure sharing to the other representation is that future reference to the constructed terms is slowed down. However, this loss of speed is quite small and amply repaid by the savings of space and the speed of creating and discarding new data structures.

In our implementation the constructed term representing the value of a variable is stored in a VARIABLE CELL consisting of the two fields :

- 1) binding ... a pointer to the source term
- 2) frame ... index of the cell/cells in which the values of the variables occurring in the source term may be found.

The variable cells for all the variables in a clause are placed together in a frame on a variable stack. Relative position of a variable within the frame is given by its offset.

The mapping between the value a variable is bound to and the contents of the variable cell is :

<u>value of variable</u>	<u>binding</u>	<u>frame</u>
undefined	to a location containing the code for undefined type.	NA
atom	to the relevant atom.	NA
integer	to the relevant integer.	NA
variable	to a location containing the code for variable type.	index of the relevant variable cell
skeleton	to relevant skeleton.	base of the relevant frame
null list	NULL	NA

At the time of creation all the variable cells in a frame have undefined values. It is only through unification that bindings are made.

Occur Check An interesting situation develops if a variable gets bound to a skeleton containing that very variable. For example consider the binding of Y in the following figure :

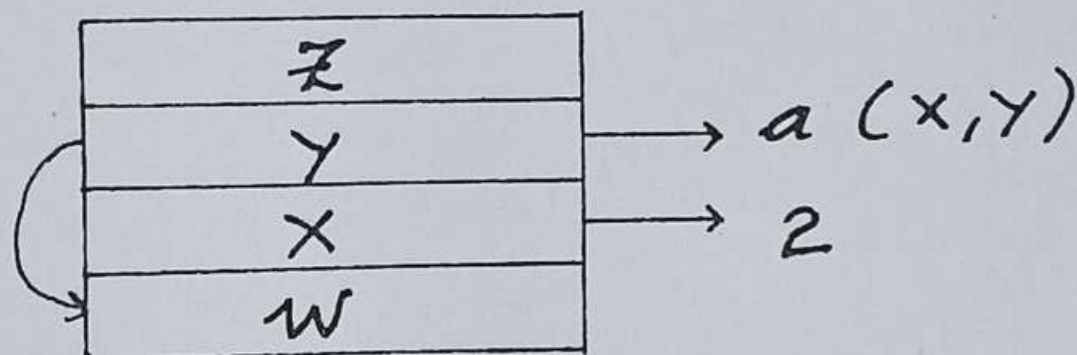


Figure 7.1(f)

The constructed term represented by Y is

$$a(2, (2, (2, (2, \dots) \dots)) \dots)$$

which is an infinite term. The inadvertent creation of such infinite terms is undesirable. A check at the time of binding a variable may be done to avoid this. However this check will have to take care of both direct and indirect circularity. Such a check is called occur check.

However, infinite terms seldom arise in normal Prolog programs, and when they do, the user may well be wanting to construct an infinite term as a valid data object. Considering this, and the fact that the check is too computation intensive, we have not implemented the occur check.

7.2 THE UNIFICATION ALGORITHM

Unification is a general purpose scheme to pattern match two terms. As a side effect the variables occurring in the terms also get bound, maybe to newly constructed terms.

Each of the two terms to be unified may be an atom, an integer, a variable or a functor. Unify succeeds if the two terms can be pattern matched. Otherwise it fails.

We will explain the unification algorithm through examples. Each example shows the two terms to be unified, the result, and if required, the steps and explanations involved.

Ex.	river	river
	success	
Ex.	river	riviera
	failure	
Ex.	river	3
	failure	
Ex.	10	10
	success	
Ex.	river	X

Here the unifier needs some more information before it can decide whether or not the atom 'river' can unify with the variable 'X'. Specifically it needs to know the binding of X, and for that it needs to know the location of the variable cell for X on the variable stack.

Success results if X is undefined (as a side effect X will get bound to the atom 'river'), or is bound to the atom 'river', or is bound to a variable which satisfies any of these criteria. In all other cases unification fails and the binding of X is not affected.

This definition of success is recursive and the chain of references from the variable needs to be traversed before one can decide on the result of unification. The process of traversing a reference chain to the last variable on that chain (which will either be undefined or bound to a non variable term) is called dereferencing. If this last variable is undefined, the result of dereferencing is a reference to its variable cell. Otherwise the

result is the term to which the last variable is bound.

For example consider the bindings shown alongside. The result of dereferencing X is the variable Z at location 4. If Z was bound to the atom 'river' the result would have been the atom 'river'.

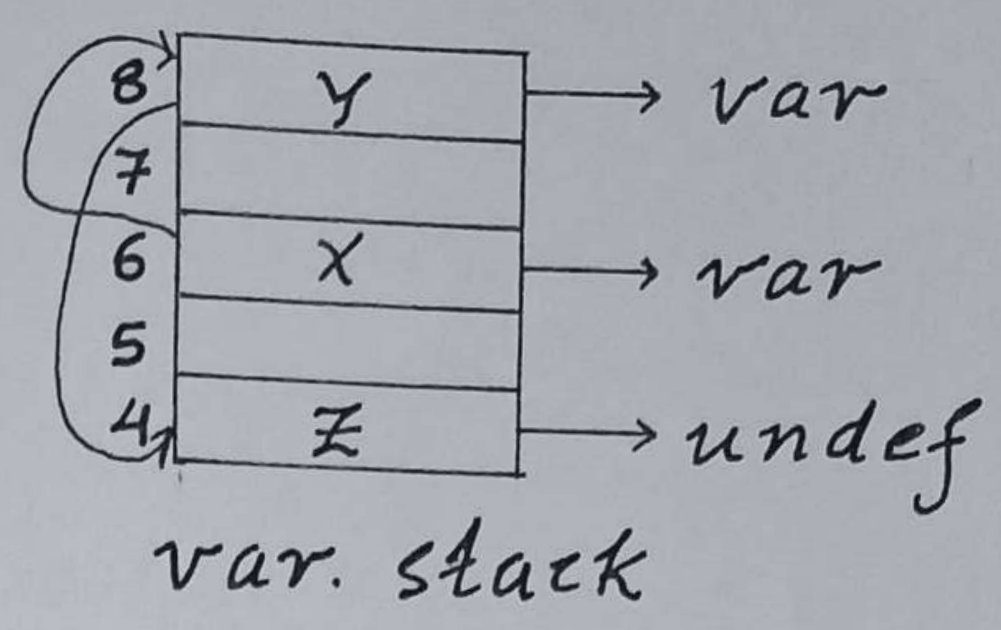


Figure 7.2(a)

As should be clear, whenever we attempt to unify a variable with another term the result is exactly the same as if the dereferenced value of the variable was used. Dereferencing of a variable before unification may be done in the unify procedure, or it may be done before calling unify. The latter scheme will guarantee that a variable sent to unify as an argument must be undefined, as otherwise its binding would have been passed. For example if the variable X, bound as in the above figure, were to be unified with the atom 'river', unify would be called with term1 = 'river' and term2 = variable at location 4. As a result of unification the variable at location 4 will get bound to atom 'river'. In case Z had been bound to 'river' in the above figure, the second argument to unify would also have been 'river'.

We chose the latter strategy in our implementation.

In all further examples we assume that terms are fully dereferenced before calling unify.

Ex. var at location 4 var at 11

success. A reference from var_stack[11] to var_stack[4] is created as a side effect.

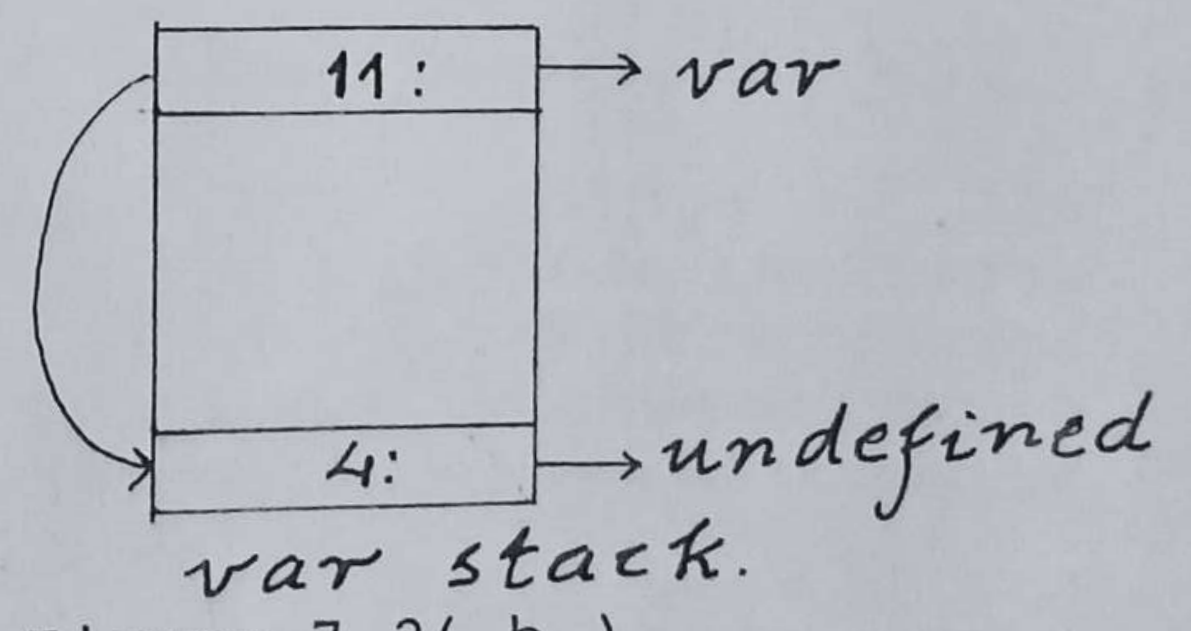


Figure 7.2(b)

The binding could as well have been from locn 4 to 11. However we always prefer downward references for reasons of efficient backtracking. This will be made clear in chapter 8.

When a variable cell is bound to a variable, the binding pointer does not point to the relevant variable record in the database. It just points to a dummy location which indicates that a variable is being referenced. The frame field of the cell contains all the information about the binding. Similarly, if a

the second pair of arguments. If that also succeeds, unification succeeds.

As should be clear by now, unification requires 4 arguments, named term1, frame1, term2 and frame2. term1 and term2 are pointers to the terms to be unified. The interpretation of the frame pointer depends on the nature of term. For a compound term, frame gives the index of the base of the frame of the containing clause. For a variable term, frame gives the index of the variable's cell. For all other types of terms, frame is meaningless. This information is summarized in the following table :

<u>Nature of term</u>	<u>Arguments to unify</u>
atom	term : ptr to atom in database. frame : not applicable.
integer	term : ptr to integer in database. frame : not applicable.
skeleton	term : ptr to functor in database. frame: index of base of frame for the containing clause.
variable	term : ptr to a dummy location indicating that the term is a variable. frame : index of variable's cell on the variable stack.
null list	term : NULL . frame : not applicable.

We will now give the unification algorithm and then discuss a detailed example.

```
unify( term1, frame1, term2, frame2 )
```

The following global data is accessed :

```

var_stack[]
vs_old_top
trail_stack[]

```

- ... each element is a variable cell.
- ... location upto where the variable stack is automatically compressed when backtracking attempts to undo the effects of this unification.
- ... references to variable cells bound during this unification are placed on this stack unless they will be automatically removed due to compression of the variable stack on backtracking.


```

if( term1 EQ term2 )
  if( null_list( term1 ) OR
      isatom( term1 ) OR
      isinteger( term1 ) OR
      frame1 EQ frame2 )
    return SUCCESS ;
    /* Immediate success if the two terms are
       identical. Identical variables or functors
       must also be referenced in the same frame */
if( isatom( term1 ) AND isatom( term2 ) )
  return( term1 -> name EQ term2 -> name ) ;
if( isinteger( term1 ) AND isinteger( term2 ) )
  return( term1 -> value EQ term2 -> value ) ;

if( isfunctor( term1 ) AND isfunctor( term2 ) )
  begin
    if( term1 -> arity NOT_EQ term2 -> arity )
      return FAILURE ;
    if( term1 -> principal_term -> name
        NOT_EQ term2 -> principal_term -> name )
      return FAILURE ;
    for( i = 1 to term1 -> arity )
      begin
        t1 = term1 -> argument[i] ; t2 = term2 -> argument[i]
        fr1 = frame1 ; fr2 = frame2 ;
        if( isvariable( t1 ) )
          begin
            fr1 = frame1 + t1 -> offset ;
            fr1 = dereference( fr1 ) ;
            if( isundefined( fr1 ) )
              t1 = var ; /* dummy variable */
            else
              begin
                t1 = var_stack[fr1].binding ;
                fr1 = var_stack[fr1].frame ;
              end
          end
          Similarly dereference the second term t2 ;
          If( NOT unify( t1, fr1, t2, fr2 ) )
            return FAILURE ;
          end /* go on to unify next argument pair */
        return SUCCESS ; /* You come here if the two functors
                           have the same name and arity, and
                           if all the argument pairs unify.
                           */
      end /* of the case when both terms are functors */

if( isvariable( term1 ) AND isvariable( term2 ) )
  begin
    if( frame1 > frame2 ) /* create a downward reference */
      reference( frame1, frame2 ) ;
    else
      reference( frame2, frame1 ) ;
    return SUCCESS ;
  end

```


Unification

```
if( isvariable( term1 ) ) /* only term1 is a variable */  
  bind( frame1, term2, frame2 ) ; return SUCCESS ;  
if( isvariable( term2 ) ) /* only term2 is a variable */  
  bind( frame2, term1, frame1 ) ; return SUCCESS ;  
end /* of unify */
```

The function 'bind(fr1, t2, fr2)' binds the variable cell at location fr1 on the variable stack to the constructed term (t2, fr2). In case this variable cell won't be automatically removed on backtracking (fr1 > old_vs_top), a reference to it is placed on the trail_stack.

The unification algorithm for two dereferenced terms can be summarized as :

- 1) If none of the terms is a variable, success occurs if both terms are null lists, equal atoms, equal integers, or functors with same name, arity, and unifiable arguments.
- 2) If only one of the terms is a variable, it gets bound to the other term. Success is ensured.
- 3) If both terms are variables, the more senior reference is assigned to the more junior reference. By seniority we mean the position on the variable stack - the lower one is more senior. Success is ensured.

Whenever a cell is bound to a value, it is usually necessary to remember the binding so that it can be undone on subsequent backtracking. The exception is when the cell will in any case be discarded on backtracking. This condition can easily be detected by the fact that the cell's address will be greater than the value of vs_old_top, the base of the frame for the clause being unified with the parent goal. If this condition is not met, the assignment is remembered by pushing the address of the cell concerned on the trail stack. It is here that the referencing of junior cells to senior ones is helpful. This scheme reduces the number of bindings to be saved on the trail stack, since the chances of the junior cell being removed automatically are greater than that of the senior cell. As an added benefit, the chain of references seldom grows beyond one or two levels if this scheme is followed.

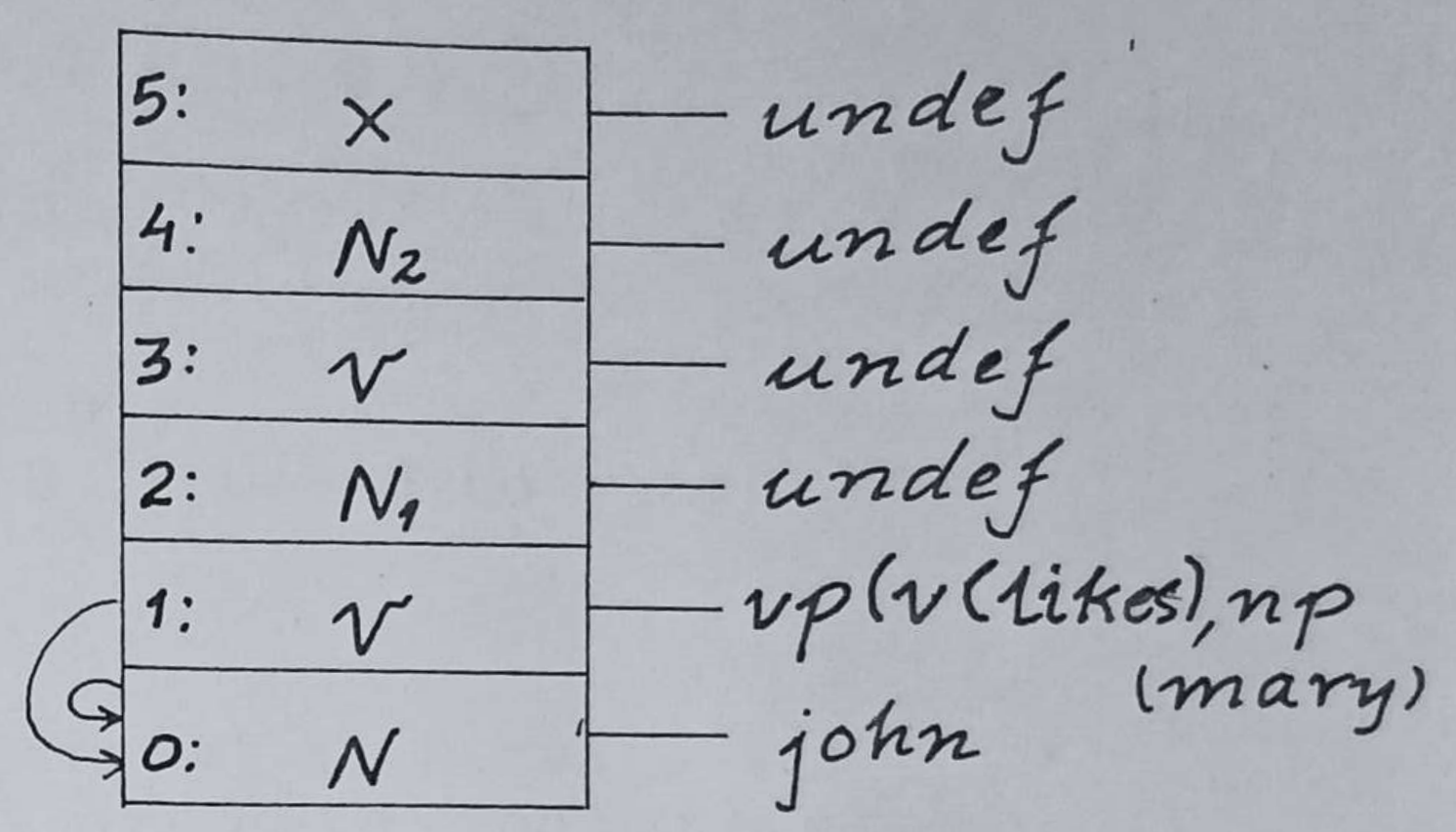
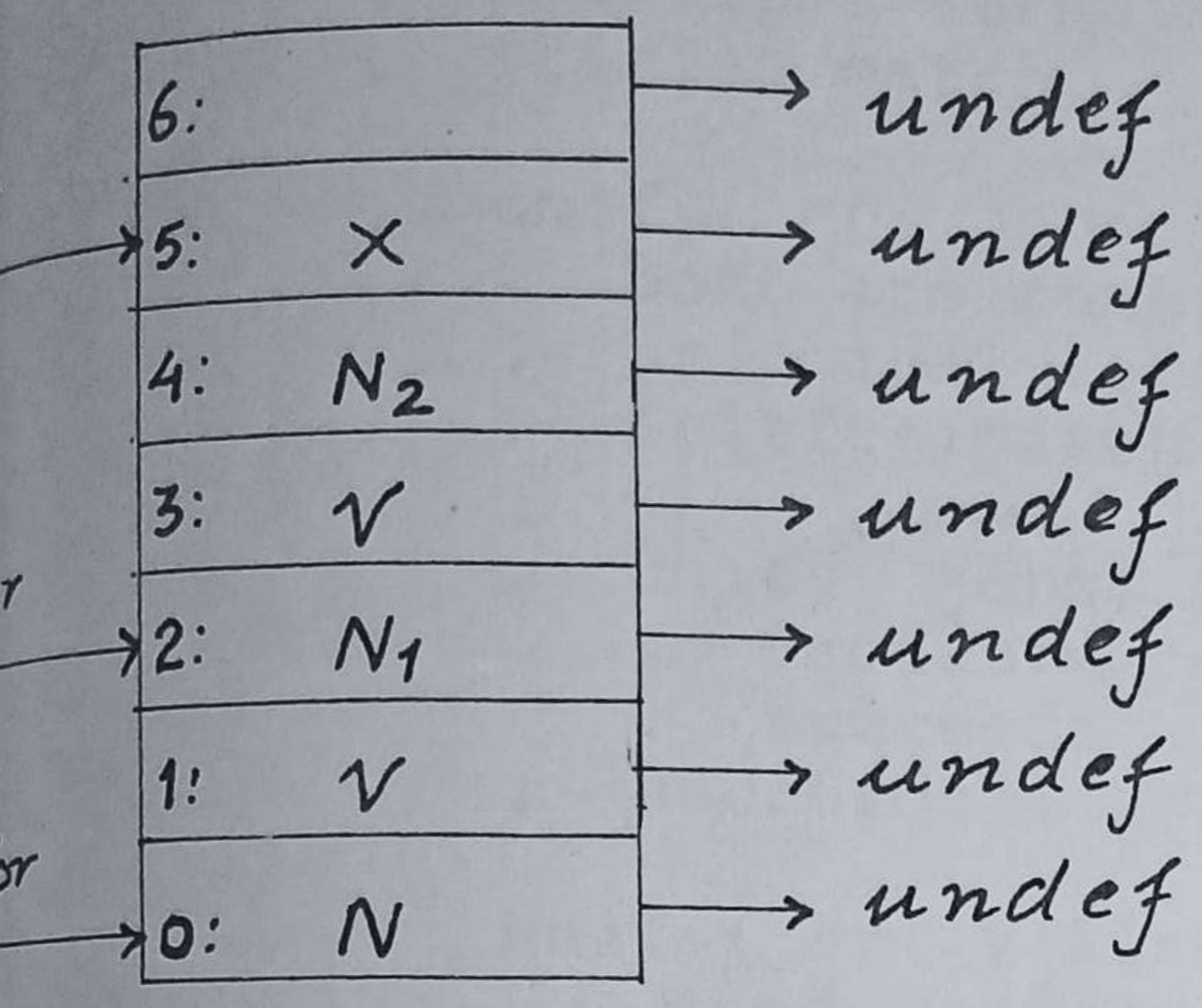
A FULL EXAMPLE OF UNIFICATION

Consider the terms :

- 1) s(np(john), vp((likes), np(mary))).
- 2) s(np(N), V).
- 3) s(N1, vp(V, N2)).

4) X.

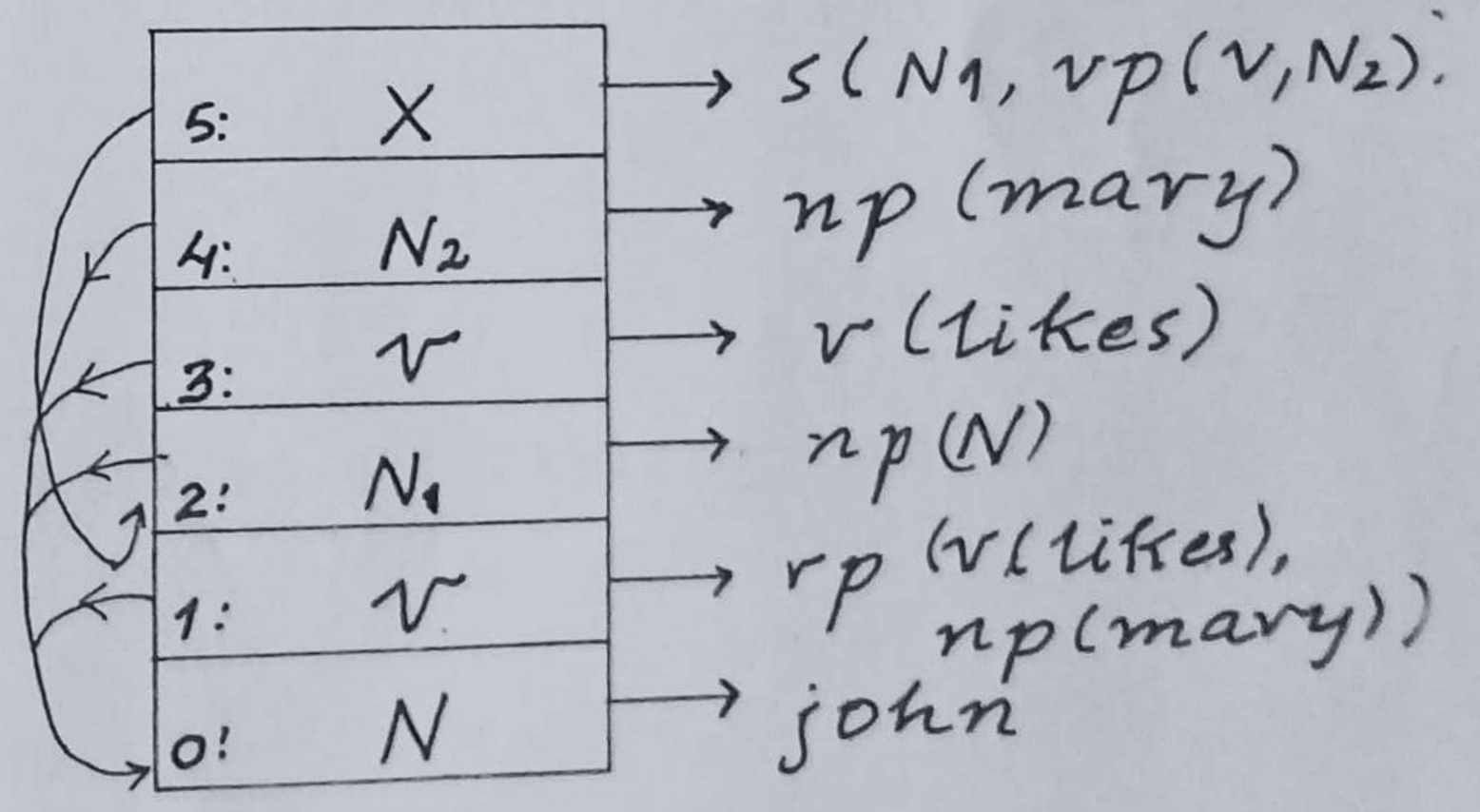
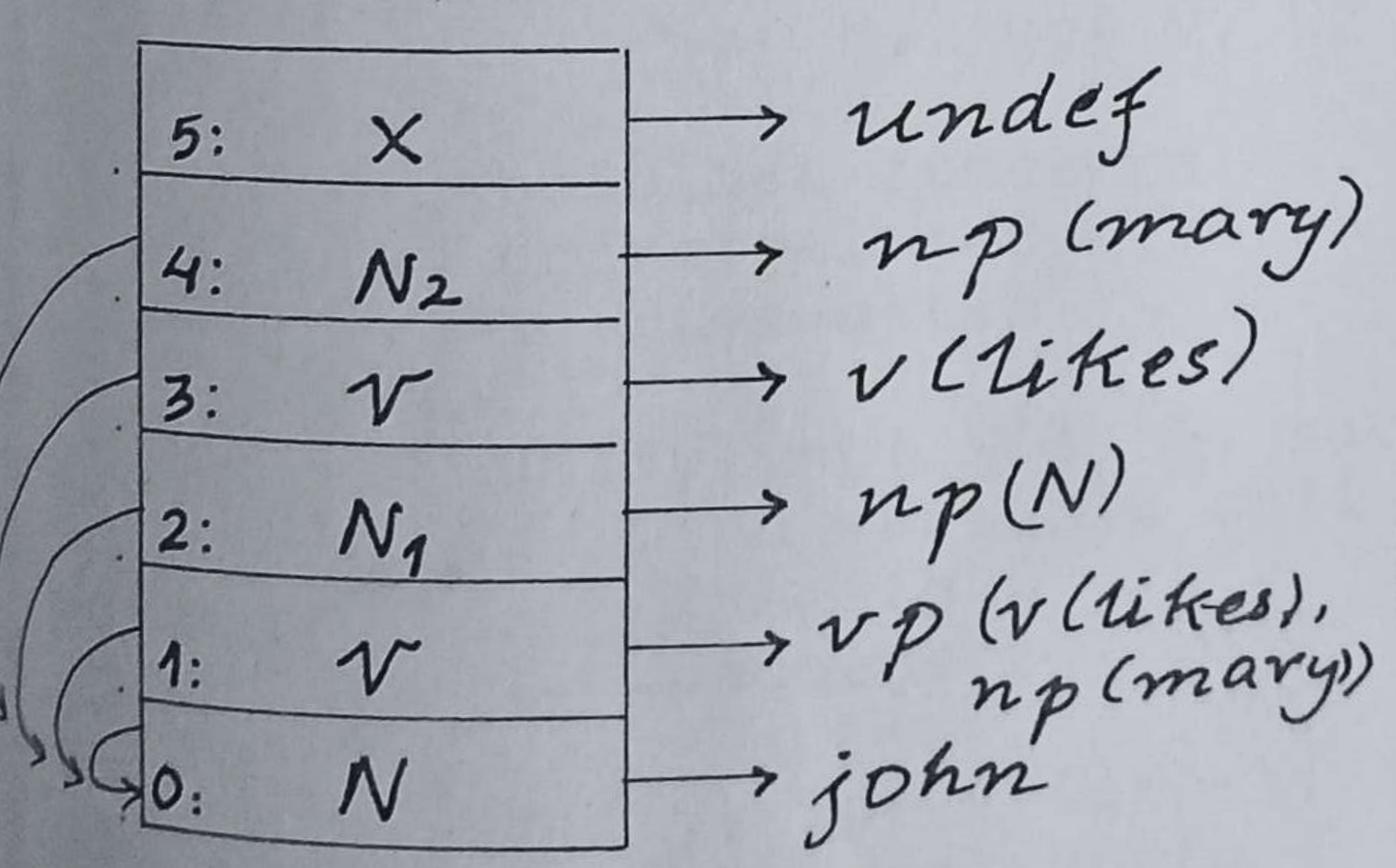
The offsets of the variables are numbered 0, 1, 2 ... depending upon their position in the clause. The frames for the terms are as shown in figure 7.2(c).



frame binding
Variable Stack.

(1) Before unification

(2) After 1st unification



(3) After 2nd unification

(4) After 3rd unification

Figure 7.2(c)

Let us consider a sequence of three unifications.

FIRST UNIFICATION

$s(np(john), vp(v(likes), np(mary))),$ at frame 0.
 with
 $s(np(N), V),$ at frame 0.

- . both are functors
- . principal terms match
- . arities match
- . $unify(np(john), 0, np(N), 0)$
 - . both are functors
 - . principal terms match
 - . arities match
- . $unify(john, 0, var, 0)$
 - . variable cell at 0 is bound to atom 'john'
 - . succeeds
- . succeeds
- . $unify(vp(v(likes), np(mary)), 0, var, 1)$
 - . bind variable cell at 1 to the skeleton $vp(v(likes), np(mary)),$ frame to 0
 - . succeeds
- . succeeds

The state of the variable stack after this unification is shown in figure 7.2(c) 2.

SECOND UNIFICATION

$s(np(N), V),$ at frame 0
 with
 $s(N1, vp(V, N2)),$ at frame 2

- . both are functors
- . principal terms match
- . arities match
- . $unify(np(N), 0, var, 2)$
 - . bind variable cell at 2 to the skeleton $np(N),$ frame to 0
 - . succeeds
- . $unify(vp(v(likes), np(mary)), 0, vp(V, N2), 2)$
 - . both are functors
 - . principal terms match
 - . arities match
- . $unify(v(likes), 0, var, 3)$
 - . bind variable cell at 3 to the functor $v(likes),$ frame to 0
 - . succeeds

CHAPTER EIGHTCONTROLLING EXECUTION : LUSH

The aim of any Prolog computation is to satisfy goals. This is done through procedure invocations wherein a goal, say G, of some goallist, say GL, is unified with the head of a clause, say C, and then an attempt is made to satisfy the body of that clause. When doing so the state of the Prolog system prior to unification needs to be conserved so that execution can return to it on successful execution of the body, and continue with the next goal on the goallist GL. This situation is exactly similar to a conventional language and is handled similarly, by a control stack. The difference lies in the non-determinancy of the execution of the goals. We may try to find an alternative solution for the same goal G later, due to failure of a subsequent goal. This is known as backtracking. Due to backtracking the status for G must remain on the control stack even after its successful execution. Backtracking may cause us to return to this status to try out other alternatives. Only after all the alternatives have been obtained (through repeated backtracking) can the status be removed from the control stack.

When Prolog backtracks to a procedure, it needs to undo the effects of the last unification. This involves unbinding any variables that got bound during that unification. The state saved must therefore include some backtracking information to enable the system to do so.

We will now describe the information that comprises the state of the Prolog system during execution of a goal. But first we clarify the terminology used.

The goal we're currently executing (attempting to satisfy) is called the current goal. The clause in which it is contained is called the current clause.

The procedure in which a match for the current goal is being found is said to have been invoked by the current goal. The current goal itself lies in a clause which was invoked by some goal. The goal which invoked the current clause is called the parent, or, activating goal of the current clause, and also of the current goal.

For example, consider the clauses :

```
likes( mary, john ).
likes( mary, charles ).
likes( jane, mary ).
likes( mary, X ) :- likes( mary, Y ), likes( Y, Z ).
```


These four clauses form the procedure for 'likes'. This procedure is invoked whenever we try to satisfy a 'like' goal, such as :

?- likes(Who, mary).

When this goal unifies with 'likes(mary, X)' it is said to invoke this clause, and to be the parent of this clause, as also of any goals within it.

8.1 STATE OF THE PROLOG SYSTEM

The state associated with the execution of a goal includes the following fields of information :

1) GOALLIST.

A pointer to the current goal and its continuation. The continuation is the list of goals to be executed after the current goal is satisfied.

2) FRAME1

Gives the base of the frame on the variable stack associated with the current clause. The frame contains variable cells for each variable in the current clause, and hence in the current goal.

3) PARENT

Pointer to the parent environment on the control stack. Execution must return to the parent goal's continuation when the current clause succeeds.

4) CLIST

A pointer to the currently invoked clause. Since a clause record contains a next clause pointer, clist effectively points to the list of clauses remaining in the invoked procedure. When saved on stack, clist would point to the clause last matched with the current goal. Backtracking would cause the system to search for alternative solutions starting from the clause

clist -> next_clause.

5) FRAME2

Gives the base of the frame on the variable stack for the invoked clause (one pointed to by clist). At time of backtracking, the variable stack is cut back till frame2. The value of frame2 is also made available to unifier and predefined predicates through vs_old_top, so that any bindings to variable cells below it can be remembered on the trail.

6) TRAIL POINTER

Gives the top of the trail stack as it stood before unification of the current goal with the currently

invoked clause head. All entries on the trail stack above trail_pointer are made during or subsequent to the current unification. Hence they all need to be undone when backtracking to this state.

7) BACKTRACK ENV

A pointer to the environment on the variable stack to which the system must backtrack if and when the current goal fails. This is the most recent environment, preceding the current one, for which the clause activated was not the only remaining alternative for the activating goal.

The last four fields of the state are required only for backtracking.

8.2 THE LUSH ALGORITHM

Our control program is based on the lush system of Kowalski (8). The logical background to this system is discussed by Van Emden (7). We first discuss the algorithm as implemented. An example is then given remove any doubts that may linger on after reading the algorithm.

The current environment of the Prolog system is maintained in the variables just discussed. Further, for each goal that unifies with a clause head, the environment before unification is stored in a control record on the control stack, where it remains for later use during a successful return or backtracking. This control record is removed only when the goal fails because there is no other clause to match it.

The data structures used are :

- control_stack[] ... each element has the seven fields required to store the state of the system
- goallist, frame1, parent, clist, frame2, trail_ptr, backtrack_env ... the seven fields of the current state
- env ... top of control stack
- var_stack[] ... top of variable stack
- vs_top ... saves vs_top for shallow backtracking, is mostly identical to frame2
- vs_old_top
- trail_stack[] ... top of trail stack
- ts_top ... saves ts_top for shallow backtracking
- ts_old_top

Controlling Execution : lush

```
lush( query )
    /* All stack tops at a location above the last entry */
begin
    /* Initialize */
    frame2 = vs_old_top = vs_top = 0 ;
    ts_old_top = ts_top = 0 ;
    env = 0 ;
    backtrack_env = parent = -1 ;
    create a frame on variable stack for query -> numvar variables;
    goallist = query -> goallist ;

NEW_CLAUSE : /* execute goallist in a new clause */
    frame1 = frame2 ; parent = env - 1 ;

NEW_GOAL : /* A new goal in the current goallist */
    if( goallist EQ NULL )
        goto SUCCEED ;
    goal = goallist -> goal ;
    clist = goal -> principal_term_ptr -> procptr;
    /* first clause in the procedure for the goal's
       predicate. */

BACKTRACK_POINT : /* set shallow backtracking points */
    frame2 = vs_old_top = vs_top ;
    ts_old_top = ts_top ;

ALTERNATIVE : /* Try the next alternative clause in the
                invoked procedure */
    if( clist EQ NULL )
        goto FAIL ;
    create a frame for clist -> numvar variables ;
    if( unify( clist -> head, frame2, goal, frame1 ) )
        begin
            if( clist -> next_clause EQ NULL )
                /* determinate execution : no point in backtracking to
                   this goal again */
                shrink trail stack between ts_old_top and ts_top by
                removing references to variables above frame2 of the
                backtrack environment ;
            else
                backtrack_env = env - 1 ; /* set to saved environment */
        end
    else /* fail to unify : shallow backtrack */
        begin
            vs_top = cs_old_top ;
            clear bindings made during fauiled unification, use
            references on trail stack above ts_old_tp ;
            ts_top = ts_old_tp ;
            clist = clist -> next_clause ; /* next alternative */
            goto ALTERNATIVE ;
        end
end
```

... Code continues on next page


```
FAIL :
  if( backtrack env < 0 )
    report failure of query ; return ;
  else
    begin
      pop backtrack environment ;
      env = backtrack env ;
      vs_top = frame2 ;
      clear trail above trail_ptr retrieved from the
      backtrack env ;
      clist = clist -> next_clause ;
      goto BACKTRACK_POINT ;
    end

SUCCESS : /* parent goal succeeds : next satisfy its
           continuation */
  if( parent >= 0 )
    begin
      restore parent environment's frame1, goallist and
      parent ;
      goallist = goallist -> next_goal ;
      goto NEW_GOAL ;
    end
  else
    begin
      print variables in query ;
      if( more solutions desired )
        goto FAIL ;
      else
        return ;
    end
  end
```

8.3 DISCUSSION OF THE ALGORITHM

Lush is written as a finite state machine with each state being represented by a label.

Lush initializes itself by clearing all stacks, creating a frame for the query, and setting the query's goallist to be the goallist of a new clause to be satisfied. Since a failure or success of this goallist represents a failure or success of lush, both parent and backtrack_env are set to invalid values (-1).

When attempting to satisfy a new clause, the system assumes that the head of this new clause was just unified with the parent goal, the parent environment was saved, and the clause' goallist made the current goallist.

After preparing to process a new clause, the next goal in its goallist must be selected. At this point, the interpreter must also find the list of clauses which may match the goal. For example, if the goal is :

Controlling Execution : lush

append([1,2], [3], X)

lush finds the clauses which describe 'append'.

Once these clauses have been found, lush will attempt to unify the goal with the head of each clause until a unification succeeds. There are several things to note at this point. Since the unification algorithm will try to match the current goal with the head of the clause, the system must have pointers to two frames : the frame for the goal, frame1 and the frame for the head, frame2.

If unification fails lush will try to match the next clause in the list. When this happens, the environment must reset to its state before the failed unification began. When this is done, the alternative clause is attempted. This is called shallow backtracking. The control stack is not affected, since the information required for shallow backtracking is stored in the variables old_sp and old_tp.

If unification on all the clauses in a given procedure fails then the system must perform deep backtracking. This is done by popping the backtrack environment from the control stack and using it to restore the environment of an earlier state in the computation.

An environment is saved when a goal successfully matches the head of a clause. The information stored in the control stack record consists of parameters already discussed.

Suppose we execute the clause :

P :- Q, R, S.

Also suppose that Q has been executed successfully and Prolog now attempts to execute R. Note that when the system found a match for Q it recorded the environment before the match on the control stack. If R fails completely, then Prolog can return to Q and retry it, hoping that the alternative solution for Q will enable R to be satisfied.

Restoration of the backtrack environment takes place in the 'FAIL' state. After that Prolog goes to the 'BACKTRACK POINT' to restart computation.

Once unification has been successful, and the environment saved, Prolog begins processing the clause which matched the goal. This is done very simply. Since the calling or parent environment has been saved on the control stack, the system can reassign the 'variable goallist' to the vector starting with the first goal in the new clause. The frame pointers and the parent must also be updated.

An entire clause has been executed successfully when all the goals in the clause have been satisfied. This condition is

Controlling Execution : lush

recognised when goallist, which is used to scan along the goals in a clause, becomes NULL.

When a clause has succeeded, control should return to the parent clause. The variable 'parent' points to the parent's environment record on the control stack. Thus lush can reconstruct the calling environment and continue executing the parent clause.

Note the difference between backtracking and returning to a parent. In the previous example, when R failed, records on the control stack were removed until Prolog could find an environment in which alternative solutions existed. If R succeeds, the parent's environment is restored without removing any records put on the stack during the execution of R. So if S fails, the computation of R can be restarted.

The backtrack environment is usually advanced at the time of saving an environment. There is just one exception. If there are no other clauses to match the saved goal, the goal can't have any alternative solution. Hence it makes no sense to backtrack to this goal later. The backtrack env is therefore retained at the last environment for which an alternative clause existed. This allows recovery of some space from the trail stack also. References to the variable cells below frame2 that got bound during the current unification are stored on the trail stack. Now on backtracking the variable stack will anyway be cut back till backtracking environment's frame2, which is less than the current frame2. Any newly pushed references to cells between these two locations can be removed.

Once a parent environment has been restored, the parent clause luses again. The pointer 'goallist' is advanced to the next goal pointer and the procedure goes to the 'NEW_GOAL' state.

If a clause has succeeded, and it has no parent (i.e. 'parent' points to the base of the control stack) then Prolog has returned to the original query by the programmer. Thus a complete solution has been found. At this point, the system can print the values of the unbound variables given in the query.

There may still be other possible solutions. These can be found by trying to match all possible alternative clauses left in the control stack. Remember, the environment records which store the alternative clause lists are only removed on backtracking. After printing one solution, if more solutions are desired, lush goes to the 'FAIL' state and begins backtracking to find the next solution. The system will continue to do this until all combinations of clause matches have been tried. When 'env' reaches the base of the control stack there are no more alternatives left and lush falls out of the bottom of the procedure and ends. Thus we can view the execution of the Prolog program as a depth first search through the space of all possible solutions to a problem.

8.4 EXAMPLE

Given the following procedure for **append**

```
append( [], L, L ).  
append( [X|L1],L2, [X|L3] ) :- append( L1, L2, L3 ).
```

consider the execution of the query :

```
?- append( [ 1, 2 ], [ 3, 4 ], X ).
```

The query is satisfied after three recursive calls to the procedure **append**. We discuss each call in turn, and give a trace of the stacks (control, variable and trail) after each call.

FIRST CALL

```
goallist = append( [ 1, 2 ], [ 3, 4 ], X ).  
frame1 = 0      ( the frame consists of a single cell for  
                the variable X )  
frame2 = 1
```

Unification of the query goal with the first clause in **append** procedure fails since the first arguments don't match. It succeeds with the second clause causing a set of bindings. X in query gets bound to the list [X|L3], X (of the clause) gets bound to the integer 1, L1 to the list [2], L2 to the list [3, 4]. L3 remains uninstantiated, and this is the value that will be determined during the next calls to **append**. This set of bindings is shown on the variable stack.

The matched clause is the last in the **append** procedure. Hence one need not backtrack to this goal. Accordingly the value of `bcktracking_env` is left unchanged.

During unification a reference to X (of query) is pushed on the trail stack, so that this X may be unbound when backtracking to resatisfy the goal. Since the system will never backtrack to this goal, this entry is however removed soon after.

Having found a match the system stores its state, as it existed just before the unification, on the control stack. It now tries to satisfy the goallist

```
append( L1, L2, L3 )
```

appearing in the body of the matched clause. This involves another invocation of the **append** procedure.

The state of the system after the first call is diagrammed in Fig. 8.4(a).

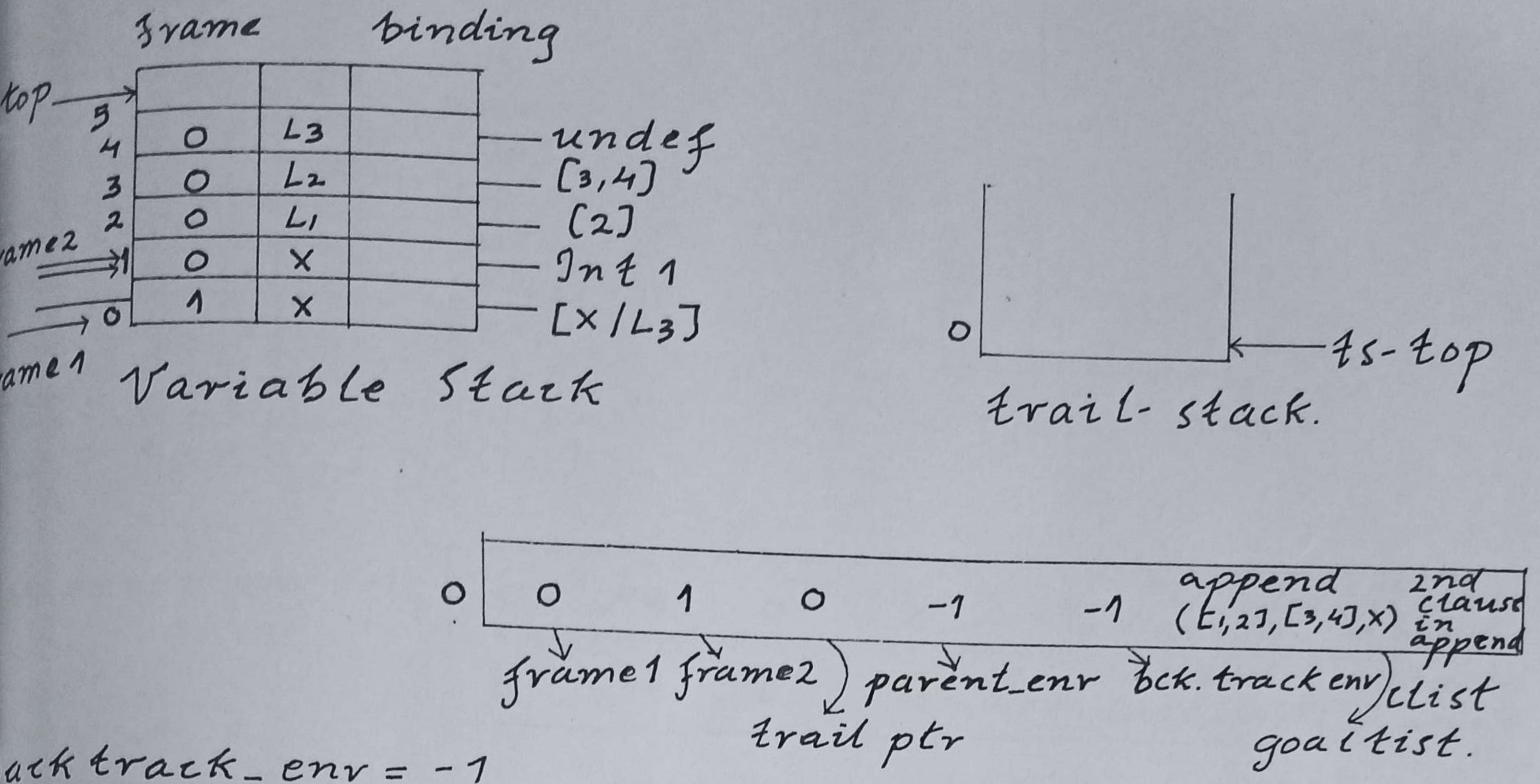


Figure 8.4(a)

SECOND CALL

The goal

append(L1, L2, L3)

with L1 = [2], L2 = [3, 4] and L3 uninstantiated, also unifies with the head of the second clause. More bindings are created as the variable stack grows. The sequence of actions is very similar to that in the first call. The state after invoking **append** again is shown in Fig. 8.4(b).

THIRD CALL

The goal

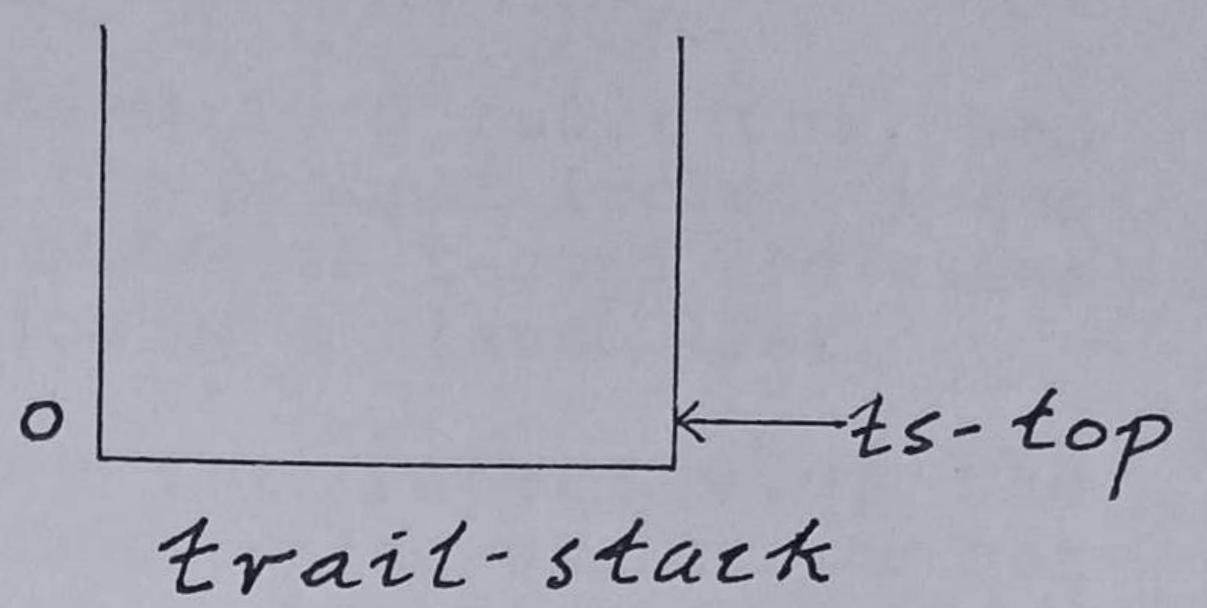
append(L1, L2, L3).

with L1 = [], L2 = [3, 4] and L3 uninstantiated, unifies with the first clause in the **append** procedure. The new frame created on the variable stack has only one cell (for L of the clause). The bindings after the unification are shown in figure 8.4(c).

This time the matched clause is not the last one in the procedure. The `bcktrack_env` is therefore updated to the newly procedure. The `bcktrack_env` is therefore updated to the newly procedure. The `trail-stack` entry made saved status of the current goal. The `trail-stack` entry made during unification is also left untouched. Fig 8.4(c) sketches the full status after this call. Lush now attempts to satisfy the (null) `goal list` in the body of the first clause.

Controlling Execution : lush

9				
8	0	L3		undef
7	1	L2		[3,4]
6	1	L1		[3]
5	1	X		Int 2
4	5	L3		[X/L3]
3	0	L2		[3,4]
2	0	L1		[2]
1	0	X		Int 1
0	1	X		[X/L3]



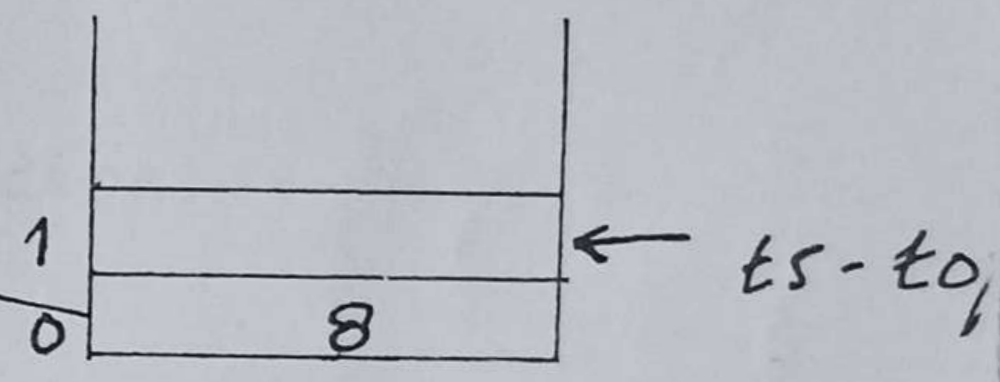
back track env = -1

1	1	5	0	0	-1	append (L1, L2, L3) 2nd clause in append
0	0	1	0	-1	-1	append ([1,2], [3,4], X) 2nd clause in append

frame binding

Figure 8.4(b)

10				
9	5	L		[3]
8	5	L3		[3]x
7	1	L2		[3,4]
6	1	L1		[3]
5	1	X		Int 2
4	5	L3		[X/L3]
3	0	L2		[3,4]
2	0	L1		[2]
1	0	X		Int 1
0	1	X		[X/L3]



back track env = -2

2	5	9	0	1	-1	append (L1, L2, L3) 1st clause in append
1	1	5	0	0	-1	append (L1, L2, L3) 2nd clause in append
0	0	1	0	-1	-1	append ([1,2], [3,4], X) 2nd clause in append

Figure 8.4(c)

The new goallist, being null, succeeds immediately. Control returns to the parent goal (3rd call), whose status is available at parent env (2) on the control stack. The next goal being null, the 3rd call succeeds, and control passes to its parent (2nd call). The process continues until the 1st call succeeds. At that time the system reads off the value of X from the variable stack and declares the query to have succeeded.

X = [1, 2, 3, 4]

8.5 INTERFACING OF PREDEFINED PREDICATES

All predefined predicates are coded as C functions, and pointers to the functions are stored in the procptr fields of the corresponding atom records. A flag in the atom record indicates whether the procptr points to a C function or a clause list.

The changes to be made in lush for interfacing the predefined predicates basically deal with checking whether or not a goal is predefined, and calling the C function if it is. Otherwise the goal is handled normally.

A predefined goal may fail, in which case the system does a deep backtrack, or it may succeed, in which case the system continues with the next goal. If the predefined predicate succeeds determinately, nothing else needs to be done, except perhaps to remove any trail references to variable cells above the backtrack environment's frame2. However, if the predicate has non-determinate execution, the status of the current goal needs to be saved, and backtrack env enhanced. Since the performance of such predicates may, in general, depend upon the number of time the goal is invoked during backtrack, the call number needs to be included as a parameter to the C function for the predicate. This is handled by using 'clist' to store the call number (instead of a pointer to a clause list) whenever a goal is predefined.

Thus, the C functions for predefined predicates have 3 arguments :

1) GOAL

Pointer to functor record of the goal to be satisfied.

2) FRAME

Base of the frame in which values of variables appearing in the goal may be found.

3) CALL NUMBER

The number of time the predicate is being called from a particular goal. The value is 1 for the first call, 2 for the first backtracking call, 3 for the next backtracking call, and so on. The parameter is meaningful for non-determinate predicates only.

CHAPTER NINE

THE CUT OPERATION

Cut is implemented through a C function like any other predefined predicate. Three important actions are performed on encountering a cut in a list of goals :

9.1 RESET BACKTRACK ENVIRONMENT

Cut semantics demand that backtracking to the left of the cut goal cause failure of all goals upto, and including, the parent goal. This means that when Prolog backtracks to the left of cut, its behaviour is exactly as if the parent goal has failed. Therefore on encountering a cut in a list of goals, the backtracking environment needs to be reset to the backtracking environment of the parent. The later can be recovered from the control stack.

```
bcktrack_env = control_stack[parent_env].bcktrack_env ;
```

9.2 SHRINK TRAIL STACK

Since the next backtrack is to occur to a much senior environment, all trail references to cells above frame2 of the new backtrack environment can be removed. Only the entries made since invocation of the parent goal need to be checked for these references as the new backtrack environment was already the backtrack environment for the parent goal.

9.3 RECOVER CONTROL STACK SPACE

The next backtrack is to the new backtrack environment which is definitely below the parent environment on the control stack. The next successful return is to the parent environment. Therefore all control records above the parent's environment are useless. They will never be referred to again and can be thrown away. Thus the control stack space upto the parent's environment can be recovered.

```
env = parent_env + 1 ;
```


CHAPTER TEN

THE PREDEFINED PREDICATE INTERFACE

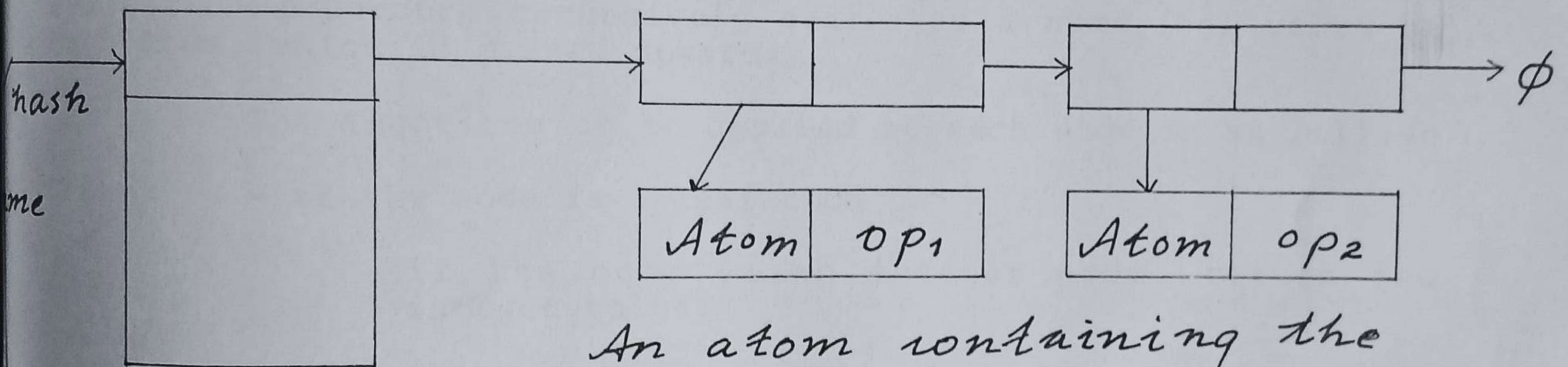
10.1 Implementation of some important predefined predicates

We discuss below the implementation of some of the more important predefined predicates.

op

op is used to create a new operator definition. It is supplied with the operator name, its specification, and its precedence as parameters.

op maintains a record of all existing operator definitions in an operator table. The table has the structure below -



An atom containing the operator name.

Each element of the operator table has a pointer to a list of operators that hash onto that index. A list element has a pointer to an atom record that stores the name of the operator. It also contains a field that describes the the precedence of the operator.

We have imposed the restriction that all operators of the same precedence must have the same specification. The array `spec_for_prec` maintains the specification of each precedence level. Multiple operators may share the same precedence level and hence the same specification. When all the operators of this precedence have been removed, it is necessary to reset the specification for that level to null. This housekeeping is done by the array `num_prec`, which stores the number of operators that share the same precedence. When, as a result of an operator deletion, `num_prec` for that precedence is also set to null.

Predefined Predicates

The op algorithm essentially is to extract the parameters of op and to place them in a newly created list element. The rop (remove op predefined predicate) algorithm is to remove the list element for the operator, and to reset spec_for_prec and num_prec as necessary.

is

'is' is the Prolog equivalent of an assignment statement. 'is' allows us to evaluate numerical operators and to carry out numerical computations.

An 'is' operator is used as in

$$Y \text{ is } X + 2 / 3$$

Note that the right hand side of the 'is' is an arithmetic expression which may contain variables to be evaluated at the time of call.

To evaluate the right hand side of an 'is' predicate, we use a recursive tree evaluation procedure. The parser creates a tree of functor records corresponding to the expression. The evaluation procedure recursively evaluates a numerical value at each node, which is passed upwards.

The algorithm to be applied at each node is as follows

- if the node is a leaf node
 - if the node is an integer node, return the integer value.
 - if the node is a variable node, then determine the binding of the variable.
 - if bound to an integer, return the integer value.
 - if bound to a structure, evaluate the structure recursively and return the value obtained.
- if the node is a tree node
 - evaluate the left and right sons recursively.
 - apply the operator specified at the node.
 - return the integer value so found.

At the topmost level, the value returned is placed in an integer record and the variable on the left hand side is instantiated to it. In case the left hand side is an integer or an already instantiated variable, then a comparison of values is

Predefined Predicates

done and success or failure appropriately returned.

The operators allowed in 'is' are + , - , / , % and * .
The % symbol stands for 'mod'.

File handling

Prolog allows files to be opened for input or output using 'see' or 'tell'. These predefined predicates maintain a current input file and a current output file. These are stored in the globals `in_file` and `out_file`. The pointers to these files are stored in `in_fp` and `out_fp`.

By default `in_fp` is `stdin`, `out_fp` is `stdout` and `in_file` as well as `out_file` are `NULL`. When a `see` or `tell` predicate is evaluated, the input and output file values are changed accordingly. Files are opened by `fopen()`, and closed by `fclose()`. Input of characters is by `getc()` for `get` and for `get0`. Output is by using `putc()`. All character input and output is in the form of ASCII integers. The characters read in are stored as integer records.

`read(X)`, used to read in a term, is implemented by parsing the term that is input and instantiating variable `X` to it. `Display` uses a set of recursive top down procedures that print out the existing clauses in the form they were input by the programmer.

consult

This predicate allows the user to read in and execute clauses and queries stored in a file. The consulted file may itself in turn consult other files, hence the predicate needs to handle this nesting of consulted files.

The method adopted is to maintain a stack of consult files. The top of the consult file stack is marked as the current file. When a call is made to consult, the current line number is saved on stack and the consulted file is pushed on the stack as the current file. Files are popped when the end of file marker is encountered. If the new top of stack element is unopened, it is opened and we start to read from line one. If the file is already opened, we resume reading from the line after the call to consult. Since line input for interpretation is always from the top element of the stack, we can automatically handle the execution of clauses and queries in the consulted file.

When the parameter for 'consult' is a list of files, then the list is pushed on stack in the inverted list order. Each file pointer on stack is set to `null(unopened file)`. The topmost file is then opened and is marked as the current file. As the end of file marker of each element of the list appears, the next file is opened and consulted.

The semantics of 'consult' are that the consultation is

predefined Predicates

initiated only after complete execution of all the goals that follow the consult goal in a clause. We would thus recommend that consult be the only goal in a clause.

delete and deletek

These are implemented by tracing the list of clauses defined by the parameter using the procedure table. The list is either unlinked, in the case of delete, or the kth element is removed, in the case of deletek.

edit

Edit allows a procedure to be output to a file for being edited. We first trace out the procedure using the procedure table. Then the procedure is listed onto a temporary file. To edit this file a system call is made to one of the standard editors. Finally the edited procedure is consulted to recreate the database records for it.

unix

The parameter to this predicate is made the parameter of a system call, thus allowing unix shell commands to be executed within the Prolog environment.

10.2 How to write your own predefined predicates

This section is meant to be a practical guide to those programmers who wish to increase the power of the Prolog system by definition of new user defined predicates.

The action defined by predefined predicates usually cannot be captured within Prolog. These actions have effects that extend beyond the Prolog system and can hence be dealt with only in the implementation language.e.g. file handling commands.

We will describe the predefined predicate interface in some detail. This is to enable future additions to the system. The motivation for this could be ease of programming by using the new predicate, an attempt to perform actions beyond the scope of the language, or an added gain in execution speed.

All predefined predicates are defined by C functions. These are placed in the file 'predef_fn.c'. Pointers to these functions are kept in a table called the 'predef_table' which is found in file predef.c .

The predef_table is actually an array of ATOM_RECORDs. The procptr field of the atom record is a pointer to the function whose name is given by the name field of the atom record. The is_predefined field for all these atom records is YES. (1).

The predef_table is initialised in the file predef.c. All predicates in the table are sorted by the name field of the atom

Predefined Predicates

record, in order to permit binary search. Hence, any new predicates must be added at the sorted position only. Further, the value of the #defined constant NO_OF_PREDEFINED_PREDICATES defined in cons_def.h should be appropriately changed. The set of predefined functions defined to be extern in predef.c should be updated to include the new function.

The function itself will be called from the interpreter with three parameters. These are

`fn_rec_ptr` : this points to the function record whose `termlist` is a list of parameters for the predefined function.

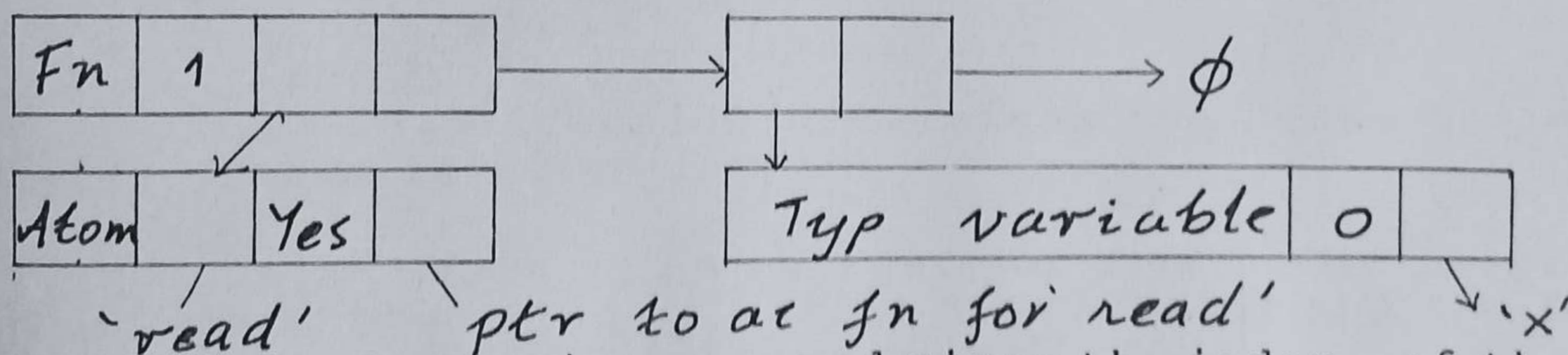
`frame` : this is the base of the stack frame for the current clause on the variable stack. The currently instantiated value of any variable can be found by looking up its cell in this frame.

`n` : this is the nth call to this function during execution of the current clause.

e.g a call to the predefined predicate 'read' will be as follows -

```
read(fn_rec, frame, n)
```

where `fn_rec` points to



frame is of type integer and gives the index of the base of the stack frame. As can be seen, the parameter for read, i.e. X is available as the first term in the `termlist` for the record pointed to by `fn_rec`.

Here are some guidelines to follow while writing predefined predicates.

First of all a thorough understanding of the structure of the database is essential since the predicate will probably need to access or modify one or the other part of the database.

There is no guarantee that the user will input any or all of the necessary parameters for the predicate, or that the parameters will be of the proper type. Hence, at every stage, extensive error handling is desirable so that in spite of programmer errors the system does not crash. In particular, before dereferencing any pointer it must be established that it

predefined Predicates

is not null.

All predefined predicates return one of three values-`DET_SUCCESS`, `NON_DET_SUCCESS` and `FAILURE`. The function returns `FAILURE` if the predicate fails. For example, the `'>'` predicate will return `FAIL` if the first argument is not greater than the second argument.

The predicate returns `DET_SUCCESS` if it has determinately succeeded. This means that on backtracking, the predicate need not be retried since the success value established will remain true even when backtracking is done across it.

The predicate returns `NON_DET_SUCCESS` if the predicate must be retried for an alternate solution when backtracking occurs across it.

Finally, as a rule, the predefined predicate must be tested on a copy of the Prolog system before it is integrated into it.

There already exist certain core functions in the file `predef_fn.c` which make the task of writing predefined predicates much simpler. These core routines extract the parameters from the `termlist`, automatically dereferencing variables and carrying out full error handling.

The core functions are

```
1: char* extr_value(fn_rec, frame, flag)
   FN_RECORD * fn_rec;
   int frame;
   int * flag;
```

This function determines the value of the first parameter in the `termlist`.

- if it is an integer, or a variable instantiated to an integer, then `flag` is set to `TYP_INTEGER` and the function returns a pointer to the integer record.

- if it an atom, or a variable instantiated to an atom, `flag` is set to `TYP_ATOM` and the function returns a pointer to the atom record.

- if it is a functor, or a variable instantiated to a functor, `flag` is set to `TYP_FUNCTOR` and the function returns a pointer to the functor record.

- if it is an unbound variable, then the `flag` is set to `TYP_UNDEF` and the function returns a pointer to the variable `cell` for the dereferenced variable.

Thus this function enables one to determine the value of the first parameter of the call. Full error handling is done automatically: if the function detects an error, it points out a runtime error and returns control to the user.

predefined Predicates

```
2: CLAUSE_RECORD * extr_clause_list(fn_rec, frame)
   FN_RECORD * fn_rec;
   int frame;
```

This returns a pointer to the clause list for the procedure whose name is given in the first parameter of the call to the predefined predicate. The function is useful in cases where one wishes to manipulate clauses whose predicate name has been passed as a parameter, e.g. in delete and deletex.

```
3: extr_clause(fn_rec, frame, i1, i2 )
   FN_RECORD * fn_rec;
   int frame;
   int *i1, *i2;
```

This looks at the first two parameters of the call and extracts their integer values, dereferencing variables if necessary. The values are returned in i1 and i2. The function is useful for defining binary predicates on integers, such as comparisons.

```
4: dereference(frame)
   int frame;
```

This is the standard function used to dereference a variable at position 'frame' on the variable stack. It returns an integer which is the dereferenced position of the integer on the stack.

```
5: run_error(message)
   char * message;
```

This is the standard function used to trap errors in the predefined predicates. A call to run_error prints the message on the standard output and jumps to the point where the processing of the next clause is initiated.

```
6: get_proc(name)
   char * name;
```

This procedure, when supplied with an atom name, returns a pointer to the procedure entry record for the procedure defined by the atom name, and if no such procedure exists, returns a NULL.

CHAPTER ELEVEN

ERROR HANDLING

The error handling strategies in our interpreter are dictated by our mode of interpretation. All input lines are processed as soon as they arrive. Hence, on any error, we are able to skip to the end of the input line (in 'panic mode'), without appreciable discomfort to the programmer.

The basic strategy to handle errors is hence to print out an error message, and then totally disregarding the rest of the input line, wait for the next line.

This strategy is well supported by the 'setjump' and 'longjump' facilities available in C. When setjump is executed, the current environment is saved. A longjump done at any subsequent stage skips all intermediate 'return' levels and returns to the point where the setjump was called, with a value specified at the point of the longjump. This enables us to terminate multiple layers of calls abruptly, and to return to the 'waiting for input' mode.

Practically speaking, a setjump is done in the user friendly interface just before the input line is expected. Later, on any error, an error message is immediately printed and a longjump done to this point. The value returned reflects the position from where the longjump was made.

The various values returned by longjumps are stored in the file 'errcode.h'. Processing is done by the user friendly interface depending upon this value.

The predictive parser table also has built in error handling: in case of a negative parser table entry, the error is automatically handled. This is done by printing the message corresponding to the negative value determined from the error message table. Subsequently, a longjump is done to the user friendly interface.

The error handling routine in the parser is 'report_error(message)'. In lush as well as predefined predicates it is 'run_error(message)' where 'message' is the error message to be printed in each case.

CHAPTER TWELVE

ORGANIZATION OF THE PROGRAM INTO FILES

12.1 INTRODUCTION

The Prolog system is spread over almost twenty different files, which are described below.

The convention is that .c files have C code, .h files are header files containing #defined variables, and .inc files contain initializations to be #included.

12.2 FILES

The files that constitute the Prolog system are -

Header files

- 1- universal.h - contains universal definitions for equality, inequality, success, failure etc. It is to be # included in all files.
- 2- deflex.h - contains definitions of terminal types used by the all parser and executor files.
- 3- defpars.h - contains definitions of non-terminals and record key types, for use by the parser.
- 4- errorcode.h - has definitions for the error status returned by a long jump.
- 5- cons_def.h - has the constant declarations that are used to control static data size e.g. the sizes of the various stacks
- 6- typedef.h - has the 'typedef' declarations for defining all the structure types used in the database records as well as in various stack elements.
- 7- include.h - has a list of all files to be #included everywhere. Thus, all one does is to #include include.h, which then automatically #includes all the required files.

Code files

- 1- lex.c - has the code for the lexical analyser.

File Organization

- 2- memman.c - has all the functions used to allocate space from the system. It includes functions such as get_integer_record(), get_atom_record etc., which allocate the database record types.
- 3- hash.c - has a single function, which hashes a character name onto a space of width 'size'. It is used as a standard hash table lookup function.
- 4- parser.c - contains code for all the six parsers.
- 5- ufi.c - has the user friendly interface.
- 6- free.c - contains the functions that recursively free space used by a database record.
- 7- print.c - has functions used to print out a database record to help in debugging.
- 8- list.c - has functions used to list out a clause in the form it was input by the programmer.
- 9- predef_fn.c - has the code for the predefined predicates.
- 10- predef.c - has the predefined predicate table as well as the necessary extern declarations for the predefined predicate functions.

Included files

- 1- fndef.inc - has the extern declarations required by the parser.
- 2- error_message.inc - has the table of error messages used by the parser.
- 3- parser_table.inc - has the predictive parsing table used by the clause, structure and list parsers.

CHAPTER THIRTEENPROBLEMS FACED DURING IMPLEMENTATION

We faced several significant problems in our implementation which we discuss below.

During the design phase, the most difficult problem was that of dynamic operator declaration. This caused us much worry since none of the standard parsing strategies was of any use. However, after consultations with our project guide, we developed a stack based adhoc mechanism that handles the problem well. This strategy is discussed in some detail in the section dealing with the atf and term parsers.

Another problem we faced was in the creation of the database during semantic analysis. Consider the following situation. Given an input

A :- B,C

where A, B and C are atomic formulae.

When this is parsed, the datf parser is required to parse D,E. It does so by calling the atf parser for parsing B and C, then creating a new ',' type node in the goal tree. Finally, B and C are hooked up to the goaltree node.

The datf parser needs the parsed records for B and C to be placed in its operand stack. Hence, the atf parser must know that whatever records it creates must be placed on the datf's operand stack. This necessitates a parameter passing mechanism between parsers. The parameter we pass is the 'location'. This is the position at which the newly created database section is to be linked up. Thus, the datf parser calls the atf parser with the location as the top of the datf operand stack. Automatically, the atf parser will link up its records onto that location. Subsequent processing can be carried out by the datf parser as required.

A similar link up problem is faced within a parser. For example, in the list parser, one needs to know where to add the next list record. This is solved by maintaining a global which always points to that location where the next element is to be linked up. A similar global is maintained in other parsers also.

The interpreter that we use assumes a clause to be in the Horn Clausal form. Thus, it does not expect any disjunctions in the set of goals to be satisfied. However, we wished to allow users the benefit of using disjunctions. To permit this, yet to allow an efficient interpreter design, we use the concept of expansion.

The user input consisting of arbitrary conjunctions and

problems

disjunctions of goals is represented by a goal tree. The tree is used to convert the input into a disjunct of conjuncts form. Finally, each conjunct is placed in a copy of the clause record and a chain of alternate clauses is built up. Thus the semantic considerations of disjunctions are fulfilled, and the efficiency of lush (our interpreter algorithm) is also maintained. The expansion algorithm is discussed along with the clause parser.

Besides these design problems, we also faced several problems with the language, C. The most irritating one was that of variable names. C allows variables of any length, but only the first eight characters are significant. We had decided upon a convention that all global variables and function names should be fully self descriptive. This necessitated using long variable names. Because of the 8 character limitation, it became rather difficult to select suitable names. Similar problems were also faced with the preprocessor statements, which also have a limited number of significant characters.

Two problems we faced were due to bugs in the C implementation in the OMEGA 58000. The first was in references of the form

$$a \rightarrow b \rightarrow c.$$

By C convention this is equivalent to

$$(a \rightarrow b) \rightarrow c$$

However, the C compiler wrongly parsed this and this resulted in highly monstrous and unrelated run time errors. We overcame this problem simply by explicit bracketing of the references.

The second problem is fairly critical and has not yet been solved. This relates to memory management routines. C provides two basic functions for managing memory. These are malloc() and free(). Malloc returns a pointer to a free memory area, and free returns it back to the system. The problem we face is that whereas malloc works perfectly, whenever a free is done, any subsequent malloc results in a run time error. We suspect that there is a major bug in the implementation of free(). This may cause us serious problems later, when larger Prolog programs are run on the system, since free space will be left dangling, and we shall be unable to recover it. Therefore we intend to write our own memory managing routines that shall allow freeing and reuse of space. However, this has not been done so far.

CHAPTER FOURTEEN

USER MANUAL

Introduction

This manual is intended for a user familiar with Prolog, but new to this implementation. Here, we discuss some peculiarities of our implementation. We also discuss a sample Prolog session in order to acquaint new users with system features.

14.1 The User Interface

The programmer interacts with the Prolog system through Prolog's user friendly interface. This provides two major facilities -

- 1) Logging of all the correct clauses and queries entered in the current session onto a workfile specified by the programmer.
- 2) Most Prolog systems, on invocation, put the user in a mode that allows only queries to be input. Consulting files is allowed as a valid query. This allows the programmer to read in previously stored clauses or to type in new clauses from the keyboard by consulting the special file "user".

The entries in a consulted file are appended to the database created during the current session. The consulted files may in turn consult other files, creating levels of nesting. We have adopted the following nomenclature for naming the levels of nested consultation. Level 0 is the outermost level, where the input is expected from the keyboard. Nested levels have correspondingly incremented levels e.g. if the programmer consults the file "alpha" from level 0, then file "alpha" is at level 1. Any file consulted from within "alpha" would be at level 2 and so on. Normally, a Prolog system allows level 0 input to be from the keyboard alone.

We have allowed the programmer the option of initiating a level 0 consultation from a previously stored file. After consultation of this file the input is resumed from the keyboard. The motivation for this approach is to allow the programmer to store queries to which he desires an answer. This is not possible in most implementations simply because the definition of consult stipulates that queries in a consult file are to be processed, but no answer is to be printed out. The implication is

that only queries at level 0 can have their their answers printed out. Allowing a file at level 0 means that queries in that file will also have their answers printed out.

Most Prolog systems do not allow facts and rules to be input from level 0. We have allowed clauses to be input from level 0, assuming that the query

```
?- consult("user").
```

was intended.

14.2 Using the System

The programmer invokes the Prolog system by means of the command

```
prolog [- inputfile] [+ workfile]
```

This means that the level 0 input will be initially from inputfile, if specified, and correct clauses and queries will be placed in workfile, if specified. The default inputfile is NULL and the default workfile is "wkfile.prlg". The user may disable logging by giving a single '+'.

A few hints to programmers are in order-

1- Since an atom will extend over all symbols till a 'white space' character is reached, all operators must be followed by a space. Otherwise, the succeeding symbols will also be considered to be a part of the operator, causing much cribbing by the parser. For example 'Y is 2*(2)' will be wrong, it should be given as ' Y is 2* (2)'.

2- Always remember to start a query with a '?-', else the query will be converted to a fact, and will be stored in the database. The next time you pose the same query, you will get a YES trivially, and will be forced to edit the clause to remove the offending fact.

3- If the system takes an inordinately long time to respond to a query, most probably the query is infinite. You should then wait patiently as penance, till the control stack space runs out and a run time error is printed. The stack is then reinitialised, so you may start afresh without any problems.

4- If you had saved the session on the standard log file, i.e wk file.prlg, then you must rename this file immediately after the session in order to save it. Else this will be erased the next time you work in prolog with default log file.

5- In case you have hit a particularly noxious bug, and

desperately need to remedy it, a copy of the listing will be available with Dr. Niraj Sharma, and in the departmental library. Otherwise try contacting us (the authors) c/o Dept. of Computer Science, UC Berkeley, CA 94720, USA for S.Keshav and c/o Dept. of Computer Science, Stanford University, CA 94305, USA for I.S.Mumick, and we will try our best to help you.

14.3 A Sample Session

We start with the initiation of the session with the command

```
prolog -likes.prlg +log.prlg
```

The Prolog system will now read in the file likes.prlg, create the database specified by the clauses in this file and also print out answers to queries in the file. As the clauses are read in, they are also printed out on the terminal.

```
likes(mary,john).
likes(john,charles).
likes(X,Y) :- likes(X,A), likes(A,Z).
?- likes(mary,X).
X = john
```

The system now waits for a response from the programmer. If the programmer wants alternate solutions, he must type in ';', else to suspend execution, he must type in '.'.

```
;
X = charles
.
```

Assuming that the file likes.prlg is now completely processed, the system now displays its standard prompt.

```
>
```

The programmer can now add clauses to the existing database and can also make queries on the database read in.

```
> likes(charles,mary).
> ?- likes(mary,X).
> X = john
>;
> X = charles
>;
> X = mary
>.
```

The programmer can also consult any other Prolog file. e.g

```
> ?- consult ( 'dislikes.prlg' ).
```

The clauses read in will be displayed, but answers to the queries in this file will not be printed.

```
dislikes(mary,john).
```



```
paranoid(X) :- likes(X,Y), dislikes(X,Y).
?- paranoid(mary).
dislikes(john,charles).
```

End of file "dislikes.prlg"

We may now make queries on the augmented database.

```
> ?- paranoid(john)
> YES
>.
```

The session is terminated by the predefined predicate 'quit' or by the end of file character **Az**.

```
>?- quit().
```

The system replies with-

```
End of Prolog session.
GOODBYE
```

At the end of the session, the workfile, consisting of the correct clauses and queries typed in will be -

```
likes(mary,john).
likes(john,charles).
likes(X,Y) :- likes(X,A) , likes(A,Y).
likes(charles,mary).
?- likes(mary,X).
?- consult("dislikes.prlg").
?- paranoid(john).
```

14.4 Modifications to Clocksin and Mellish

- 1- Consult cannot be specified as a list of files in the list format, though a sequence of comma seperated files can be an argument to consult.
- 2- ';' and ',' are not defined as operators. This prevents the programmer from using a disjunction as an argument to a predefined predicate such as 'call' and 'not'.
- 3- '[' and ']' are used for grouping disjunctions rather than '{' and '}'.
- 4- The cons symbol for list representations is '&'and not '.'.
- 5- 'yfy' specification of operators is not allowed.
- 6- Grammar rule input notation has not been implemented.

- 7- Predicates with no arguments must be followed by '()'. e.g trace() instead of trace for the 'trace' predefined predicate.
- 8- We do not allow lists to be of the form [X, Y|Z]. This must be specified as [X,[Y|Z]].
- 9- Like the tab predefined predicate, the newline predefined predicate also takes a parameter, the number of newlines to be inserted.

14.5 Restrictions

- 1- All operators of the same precedence level must have the same specification.
- 2- Once defined, an operator cannot be used in standard structure notation.
- 3- Quoted atoms and strings may not extend over two lines.

14.6 Limits of the system

The system is bounded by finite upper limits that we feel will be hard to exceed. In case of any problem, the programmer may change the controlling constant definition in the file cons_def.h. Some of the upper bounds are -

- 1- A clause may extend over any number of lines, but each line may contain at most 320 characters.
- 2- At most 50 levels of nested consultation are allowed.
- 3- There can be at most 50 variables in a clause.

etc., etc., etc.

14.6 Available Predefined Predicates

At the time of writing, the following predefined predicates have been implemented -

op	is	see	seeing	seen	tell
telling	told	put	get	get0	read
display	atom	var	nonvar	integer	true
atomic	fail	listing	consult	<	>
=<	>=	skip	nl	rop	delete
delettek	unix	edit			

Except for the predicates discussed below, all the other predicates are discussed in Clocksin and Mellish and our implementation is exactly according to this book.

rop(X)

rop removes an operator defined by op. X must be the atom for the operator or instantiated to it.

delete(X)

delete removes all the clauses for the predicate whose name is X. X must be the atom for the name, or instantiated to it.

deletek(X,k)

This removes the kth clause in the list of clauses for the procedure whose name is X. X must be the atom for the name or instantiated to it. k must be less than or equal to the number of clauses in the procedure so far

unix('X')

This allows a unix shell command to be executed from within the Prolog system. The command X is executed.

edit(X)

This allows a procedure named by X to be edited, and the new definition of the procedure overwrites the old definition. The procedure is first put in a file, and the user has a choice of ed, spy or se to edit it. The edited procedure then replaces the old definition of the procedure in the database.

REFERENCES

- 1) A.V.Aho and J.D.Ullman: Principles of Compiler Design. Addison Wesley, 1977.
- 2) Clocksin and Mellish: Programming in Prolog. Springer Verlag 1981.
- 3) Kernighan and Ritchie: Programming in C. Wiley Eastern, 1979.
- 4) D. H. D. Warren: Implementing Prolog : Compiling Predicate Logic Programs Vol. I .
DAI Research Report No. 39, University of Edinburgh, 1977.
- 5) C. A. Sammut and R. A. Sammut: The Implementation of UNSW Prolog.
The Australian Computer Journal , Vol 15 , No 2, 1983.

Other secondary references are

- 6) M. Bruynooghe, The Memory Management of Prolog Implementations.
Logic Programming, Academic Press, 1982.
- 7) M. H. van Emden : Programming with resolution logic.
Machine Intelligence 8, 1977.
- 8) R. A. Kowalski: Predicate Logic as a Programming Language .
Proc. IFIP74, 1974.

APPENDIX 1

Below is the grammar for Prolog in an easy to understand form. However, left recursion is present, and left factoring needs to be done.

```

CLAUSE ->   ATF .
           ¶ ATF :- DATF .
           ¶ ?- DATF .

DATF  ->   CATF ; DATF
           ¶ CATF

CATF  ->   CATF , GOAL
           ¶ GOAL

GOAL  ->   ATF
           ¶ $ DATF †

ATF   ->   atom ( TERM , TERM , ... )
           ¶ LIST
           ¶ OP_CONF

TERM  ->   atom
           ¶ integer
           ¶ variable
           ¶ ATF

LIST  ->   [ TERM ¶ LIST ]
           ¶ [ TERM ¶ variable ]
           ¶ [ ]
           ¶ string
           ¶ & ( TERM , LIST )
           ¶ [ TER , TER , ... ]

OP_CONF ->  TERM op
           ¶ TERM op TERM
           ¶ op TERM

```

where op is a pseudo terminal that represents a user defined operator.

APPENDIX 1A

The grammar after removal of left recursion and left factoring is given below. The convention is that a one or two letter nonterminal is used to do left factoring and primed nonterminals are used to remove left recursion.

CLAUSE	->	ATF C
CLAUSE	->	?- DATF
C	->	.
C	->	:- DATF .
DATF	->	CATF D
D	->	epsilon
D	->	; DATF
CATF	->	GOAL CATF'
CATF'	->	, CATF
CATF'	->	epsilon
GOAL	->	ATF
GOAL	->	\$ DATF †
ATF	->	STRUCTURE
ATF	->	LIST
ATF	->	TERM op TERM
ATF	->	op TERM
ATF	->	TERM op
TERM	->	STRUCTURE
TERM	->	LIST TERM'
TERM	->	op TERM TERM'
TERM	->	integer TERM'
TERM	->	variable TERM'
TERM	->	atom TERM'
TERM	->	(TERM) TERM'
TERM'	->	op TERM TERM'
TERM'	->	op TERM'
TERM'	->	epsilon
LIST	->	string
LIST	->	& (TERM , LIST)
LIST	->	[L
L	->]
L	->	TERM M


```
M      ->  ¶ N
M      ->  , LTERMLIST ]
M      ->  ]

N      ->  variable ]
N      ->  LIST ]

LTERMLIST ->  TERM LT

LT      ->  , LTERMLIST
LT      ->  epsilon

STRUCTURE ->  atom ( STERMLIST )

STERMLIST ->  TERM S
STERMLIST ->  epsilon

S      ->  , STERMLIST
S      ->  epsilon
```


APPENDIX 2

THE PREDICTIVE PARSER TABLE

	0	1	2	3	4	5	6	7	8	9
	CLAU	C	STERM, S	S	L1ST	L	M	N	LTERM, LT	LT
Atom	1 ATFC		5 TERM, S			13 TERM M			19 TERM, LT	
Variable	1 ATFC		5 TERM, S			13 TERM M		17 VARIABLE	19 TERM, LT	
Integer	1 ATFC		5 TERM, S			13 TERM M			19 TERM, LT	
String			5 TERM, S			13 TERM M		18 LIST	19 TERM, LT	
Query	2 ?-DATA									
If		4 :- DATA								
Dot	1 ATFC	.								
Semi-colon										
Comma				7, S TERM LIST			15 2 L TERM LIST			20 LTERM LIST
Left braces										
Right braces										
Left parenthes	1 ATFC		5 TERM, S			13 TERM M			19 TERM, LT	
Right parentheses										
Left sq brackets	1 ATFC		5 TERM, S		11 [L	13 TERM M		18 LIST	19 TERM, LT	
Right sq brackets										
Vertical bar						12]	16]			21 E
Cons. E			5 TERM, S		10 .(TERM, L)		14 'N			
Dollar								18 LIST	19 TERM, LT	

APPENDIX 3

FIRST AND FOLLOW FOR THE NONTERMINALS

	FIRST	FOLLOW
CLAUSE	? - atom string [.	
0	op integer variable (\$
C	. :-	.
DATA	atom string [. op integer variable ({) ; }
D	; E) . }
CATF	atom string [. op integer variable { }	; .) }
CATF'	; E	; .)
GOAL	atom string [. op integer var { }	; ; .) }
ATF	atom string [. op integer (variable	:- ; ; . }
STERMLIST	atom string . op integer variable ()
S	E ,)
TERMLIST	atom string [. op integer variable . [. :- ; .] , op
TERM'	op E	. :- ; .] op)
LIST	string [.	:- . , ; op)
L]	:- . , ; op)
M	,]	:- , ; op)]
N	variable string [.	:- , ; op)] ;
TERMLIST	atom string . op integer variable []
-T	E ,)