Design and Implementation of the Waterloo Unix Prolog Environment

Mantis Hoi Ming Cheng
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-84-47
December 1984

Design and Implementation of the Waterloo Unix Prolog Environment

by

Mantis Hoi Ming Cheng

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1984

© Mantis H.M. Cheng 1984

Authorisation

I hereby declare that I am the sole author of this thesis.

I authorise the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Mantis Cheng

I further authorise the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Mantis Cheng

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This document describes the development of a new Prolog system on a DEC VAX-11/780‡ running under the UNIX† operation system. Modular programming in Prolog and an adaptable runtime system are some of the notable features introduced in this new implementation. It is a programming environment that is expected to provide a Prolog system for developing knowledge-based systems and studying further logic programming development. Its design is primarily concerned with the long-term objectives of portability, extendability, and adaptability. The discussion here concentrates on the rationale behind and the details of this implementation. We conclude with a few remarks on the results to-date, and with some comparisons with another popular Prolog system.

†UNIX is a Trademark of Bell Laboratories.
‡ DEC is a registered trademark of Digital Equipment Corporation. VAX is a registered trademark of Digital Equipment Corporation.

Acknowlegements

Many fruitful comments came from my supervisor Randy Goebel, whose insistence on quality and excellence provided the impetus for improvements. Spontaneous discussions with Keitaro Yukawa, Patrick Chan and Claes Harvenberg have been inspiring. David Poole and Romas Aleliunas provided much insight into the extensions of the concept of modules in Prolog. Professor Maarten H. van Emden supported spiritually and made NSERC funds available for the development of WUP. Thanks to the Waterloo Symbolic Computation Group for allowing me to use their computing facilities. Special thanks to Yoichi Yano for his kindness of making available his own personal computer, Macintosh†, which was used to draw many of the diagrams and tables for this documentation.

† Macintosh is a trademark licensed to Apple Computer, Inc.

Table of Contents

List of Figures

To my beloved father and mother

Chapter 1
Overview of Prolog

Since the seventeenth century, logicians and mathematicians have studied mechanical theorem-proving to find a general decision procedure for proving theorems. With the advent of digital computers, it then became an important subject of research in Artificial Intelligence. The foundation of mechanical theorem-proving was developed by Herbrand in 1930. A major breakthrough in 1965 was the development of Resolution Principle by Robinson [Rob65]. Since then, many variations of this principle have been proposed as the basis for automatic theorem-provers.

Robert Kowalski and Alain Colmerauer invented logic programming. PROLOG (PROgramming in LOGic), a theorem-prover based on the Resolution Principle, was devised around 1972 by Alain Colmerauer and others at University of Marseilles, France [Col73]. It was first conceived as a programming tool for constructing natural language processing systems. The first experimental Prolog interpreter was implemented in Algol-W by Philip Roussel [Rou72]. The first widely used implementation was written in FORTRAN by Battani and Meloni [Bat73].

During the past ten years, techniques have been refined to provide an efficient implementation of Prolog. The DEC-10 Prolog interpreter and compiler, written in Prolog by David Warren and others [War79], has been the most widely used Prolog implementation, and is sometimes referred to as the *de* *facto* standard Prolog. Many of its features have later been adopted in other Prolog systems. Together with the contributions by Maurice Bruynooghe [Bru76,Bru82,Bru82a], Keith Clark and Frank McCabe [Cla82], and Chris Mellish [Mel82] Prolog implementation has become rather well-understood.

1.1. Current Research and Development

In addition to natural language processing systems, Prolog has also been used successfully in applications such as symbolic integration [Ber73], plan formation [War76], computer aided building design [Mar77], compiler construction [War77], database description and querying [Gal78], and drug analysis [Dar78]. Recently, attempts have been made to develop practical knowledge-based systems using Prolog [Miz83]. For other practical purposes, there is a growing interest in defining and building Prolog machines to achieve execution efficiency. This has been further stimulated by the Japanese proposal to use Prolog as the basis for the kernel language in their Fifth Generation Computer System (FGCS) project [ICO82]. One of their final goals, within ten years time, is to provide a very large-scale high speed (1 billion LIPS†) knowledge-based inference machine.

To support such large-scale applications, many present Prolog systems must be modified or redesigned to accommodate future extensions. Most existing Prolog implementations were built for experimental purposes: to investigate the feasibility of using Prolog as another programming language. To a certain extent, all suffered from very limited sizes of application.

† Logical inference per second, 1 LIP is approximately equivalent to 100 instructions per second in present computer technology [Tre82].

Current research is predominantly focused on exploiting the inherent parallelism of logic programs [Hog82,van82,Con83]. In the case of Prolog, the major concern is with AND and OR parallelism. Difficulties commonly encountered are the synchronisation of subgoal calls, search strategies in search or proof tree, and the organisation and control of subgoal generation. As far as "pure" Prolog is concerned, its data-driven and generate-and-search nature suggest the possibility of realising Prolog machines based on Data-Flow [Tre82,Wis82] or Graph-Reduction architectures [Ona82]. All of these are mostly concerned with the execution aspect of Prolog. Another area of research that has been given much attention is the practicality of logic programming in real applications: processing large databases, constructing operating systems and building expert systems.

To support the development of large knowledge-based systems, a Prolog system must provide good program development tools and be reasonably robust. This requires a coherent and integrated programming environment. The notion of modular programming in Prolog has been recently introduced and compared with some modern programming languages [Jon80,Ben80,Egg82,Chi83,Fur83]. It must be incorporated in such a way not to violate the properties of logic programming. The primary concern of the conventional use of modules is program reusability. Meta-level control and object-oriented programming are some other motivations for using modules in Prolog [Kah82,Bow82,Sha83]. As the number of Prolog implementations increase, the task of transporting Prolog programs across systems becomes non-trivial because of the differences in syntax and system predicates. Some form of standardisation must be made to eliminate this overhead of translation [Bru83].

1.2. Prolog at Waterloo

The first Prolog interpreter developed at University of Waterloo was implemented by Grant Roberts [Rob77]. It was written entirely in IBM-370† assembly language, and runs under the VM-370/CMS operating system. Although it is an interpreter, its speed is comparable to the DEC-10 compiler. Usability and efficiency were the major objectives of its designer. Notable features include user-definable error and exception handling, user-controllable runtime environment ("self-consciousness" programming, as explained by Roberts). Its major drawbacks are lack of portability, inefficient-use of runtime storage, and inadequate debugging facilities. Nevertheless, it is still the fastest Prolog interpreter around today and fairly usable [Mos80].

Ronald Ferguson [Fer81] implemented the first Unix-based Prolog interpreter in system programming language C [Ker78]. His failure of using YACC [Joh75] to generate a parser for translating Prolog programs with Operator declarations provided a better understanding of dynamic parsing. In addition to testing the initial version of ABC algorithm [van81], this interpreter contributed much insight into the control and runtime structure, and the possibility of using a high-level system language to build a reasonably efficient Prolog system.

† IBM is a registered trademark of International Business Machines Corp.

Paul Ng [Ng82] improved the design and structure on the first version of Waterloo Unix Prolog interpreter by employing the concept of abstract data types [Lis74,Gut77]. His prime motive was to increase the modularity of Ferguson's interpreter and hence its extendability. He simplified the Prolog syntax (thus, the parser) to a Lisp-like notation. An attempt was made to define a minimal but workable set of built-in predicates in Prolog. The result was a smaller, cleaner version of Prolog interpreter running on Unix.

The last two Unix-based Prolog interpreters were designed with similar goals in mind: simplicity, portability and extendability. Despite the fact that they were never released for general use, they provided much insight to the current implementation.

1.3. Summary

All three above-mentioned interpreters suffered, as in most other Prolog systems, from lack of defining long-term objectives at the outset. The conflicting objectives of efficiency and modularity sometimes create real problems in constructing an extendable system. Because the fast-expanding VLSI (Very Large Scale Integration) technology provides hope of implementing high-level computer architectures, modularity in the design and implementation of a new Prolog system is an obvious choice over efficiency in execution. The loss in efficiency can be compensated by building a Prolog machine in hardware (as proposed in FGCS) based on a viable abstract machine definition.

A programming language is not successful just because of its simplicity or mathematical elegance. It requires a good programming environment support. Smalltalk† [Ing78], Ada‡ [DOD80] and Waterloo Port [Mal83] are examples of integrating the language into its programming environment, that is self-contained and specifically tailored for the supporting language. This is the motivation behind the current version of Waterloo Unix Prolog. Our design is based on the following long-term objectives:

(1) portable (written in a high-level language)
(2) usable (good debugging facility and programming environment)
(3) extendable (modular in design and implementation)
(4) adaptable (changeable runtime library for different applications)
(5) space-efficient (secondary storage management and tail recursion optimisation)
(6) suitable for large applications (introduces the notion of modules in Prolog, and separate compilation).

Many ideas of the current implementation are based on the experience gained in the past. The result is a fully operational and usable Prolog programming environment integrated within the Unix operating system.

In the next chapter, we give an overview of this new implementation, describe some terminology for later discussion, and take a brief look at the system layout. The following two chapters form the core of this document; they specify the details of the design and implementation. Our conclusions in the final chapter include some comparisons with other Prolog systems, and a few suggestions for future extensions and the result to date. Details of how to use this system can be found in [van84,Che84].

Smalltalk is a trademark of Xerox Corporation.
Ada is a registered trademark of the U.S.Government (Ada Joint Program Office).

Chapter 2
Waterloo Unix Prolog Environment

The current implementation (hereafter, called WUP) is the third version of Waterloo Unix Prolog system. It is not an extension to the previous versions. Except for the runtime ABC algorithm on which all three versions are based, the design and implementation are completely new. With the foresight that Prolog will be heavily used in large system or application program development, WUP is designed with the long-term objectives: extendability, adaptability and usability.

2.1. Overview of WUP

WUP consists of five major components: a preprocessor, a compiler, an interpreter, a database management system and a runtime support system (see figure 2.1). The prominent features include the support of separate compilation, adaptable runtime system and modular programming in Prolog. It can be considered as an integrated programming environment built on top of Unix.

A user can create, maintain and execute Prolog programs of reasonable size. Editing facilities are provided in WUP through the standard editors on Unix and a simple interactive line-editor. Prolog programs are compiled and maintained automatically by the database management system, which interfaces with the Unix file system. With the facility of separate compilation, one can develop large Prolog programs incrementally. For specific applications, a user can also

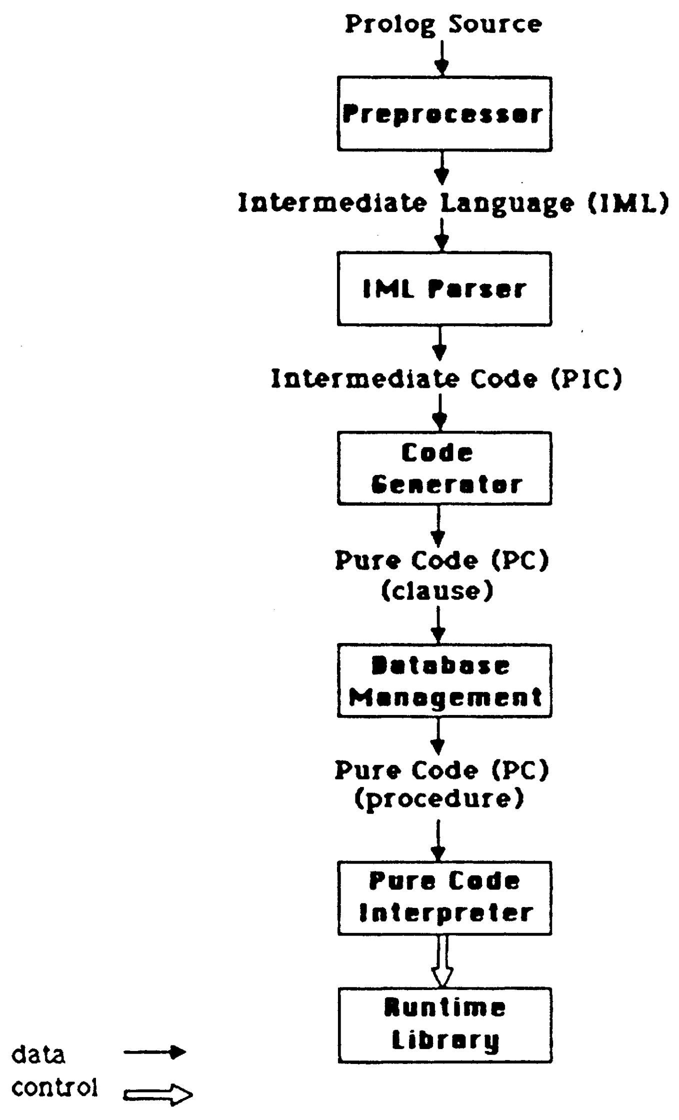


Figure 2.1 Data Flow Diagram of WUP

supply his own runtime library to replace the standard one. WUP can be modified for real application other than studying logic programming; for instance, the WUP's system primitives can be extended to include application-oriented functions like: robot-movement controls or computer graphics interface.

Details of such an interface can be found in the implementation manual [Che84]. Next, we introduce some terminology for the purpose of future discussion and give a brief outline of the overall system structure.

2.2. Terminology

A *Clause* is an assertion (a fact) or a rule as in the definition of Horn Clause (cf. Kowalski 1974). A *Predicate* is the head functor of some clause, and a *Procedure* is a sequence of clauses with the same predicate. Since WUP is integrated in the Unix operating system, the notion of *File* and *Directory* are as defined in the Unix file system.

In WUP, a Prolog program is defined by a *Program Module,* which is a directory. A program module consists of two parts: a program source and a program database. The user provides the source; WUP generates the database. The consistency between the two is automatically maintained by WUP, using a dictionary (see figure 2.2).

The program source is organised by the user as a structured collection of files under a directory. All the files (or leaves in the file structure) in the program source make up a Prolog program. A set of clauses with the same predicate name (perhaps with different numbers of arguments) must be stored in a file with the same name as the filename, a *Predicate Source File.* The corresponding program database is generated by WUP when the user invokes WUP on that program module. Each predicate source file is compiled, and the code generated is saved under a sub-directory, a *Predicate Object Directory,* with the same name inside the program database. Every program module has exactly

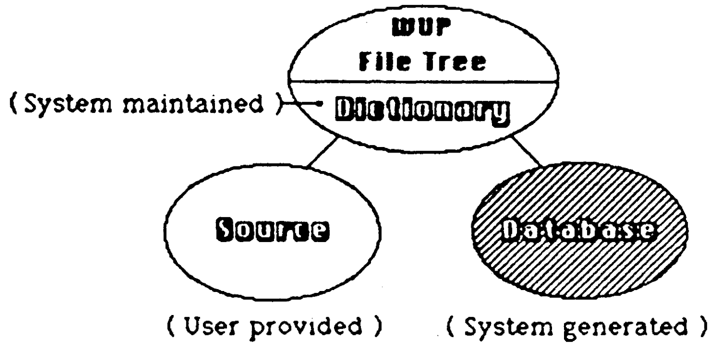


Figure 2.2 A Program Module

one program database. Subsequent modification to a predicate source file in the program source causes an automatic update on that corresponding predicate object directory in the program database. A *Dictionary* ('.dict') is maintained inside the program module to ensure the consistency between the source and the database (see figure 2.3).

A *Library* is a pre-compiled program module which can be imported and used by other program modules. In fact, the system built-in predicates and a standard set of predicates are two libraries that are included in every WUP invocation. Every library is imported with an internal user-defined *Module Name* (or *Module Alias)* that can later be referred to, and an external Unix file pathname, a *Module Path*. During a WUP session, a library can have multiple instances as long as each instance has a different internal module name. The mapping from a module name to a module path can be a many-to-one relation, but never one-to-many. The order of library inclusion specifies the default

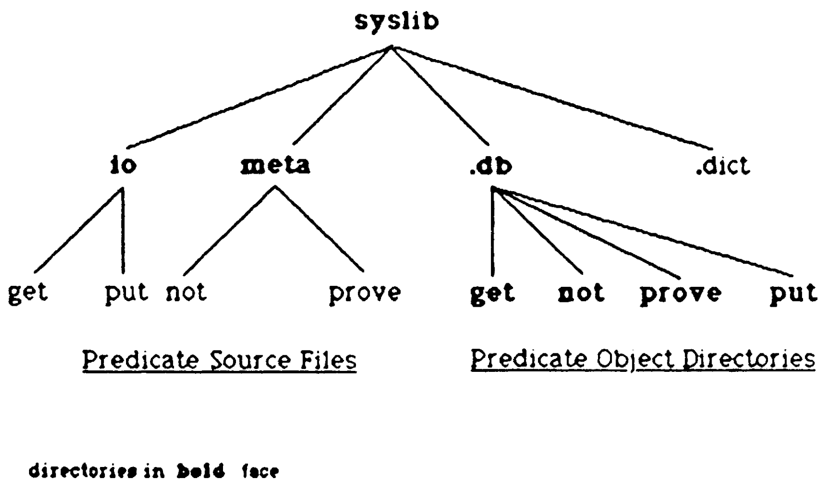


Figure 2.3 A portion of System Library File Structure

searching sequence, the *Search Path* of Prolog predicates in WUP. In figure 2.4, the module "usr" is the user's current working directory and the library "lib" is imported by the user, note that the two system libraries are always included at the end of the search path. Conflicting predicates, those with the same name and arity but are used differently, can co-exist as long as they belong to different libraries. There is nothing special about system pre-defined predicates; their definitions can be overridden by user-defined predicates.

During execution, every program module instance is represented internally by a *Module,* and its program database becomes the external database of that module. A module has three distinct parts: a user-defined name, an external path and a clause database. The clause database has three sub-databases: *Internal, External* and *Auxiliary.* (see figure 2.5). The internal database

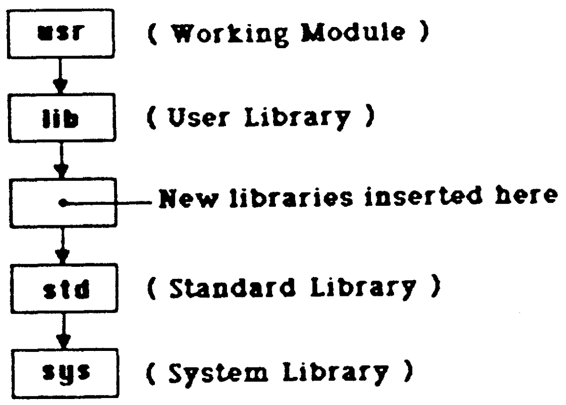


Figure 2.4 Search Path

represents the in-memory portion of a program database. The external database represents the program database on secondary storage. The internal database is initially empty and its clauses are incrementally loaded from the external database on a demand basis. At any given time, the internal database is always a subset of the external database. Together they form the read-only portion of the main database. In Prolog, there is on distinction between code and data. One can delete part of his program while it is running. The underlying Prolog system must monitor the user program to ensure the system integrity. We avoid self-modifying programs by providing auxiliary database(s) to save all clauses dynamically created during execution. A user can assert or retract clauses only in the auxiliary database. WUP ensures that the main and the auxiliary database of any module are always mutually exclusive, i.e. the same predicate cannot exist in both databases at the same time. This avoids the runtime overhead of monitoring self-modifying code.

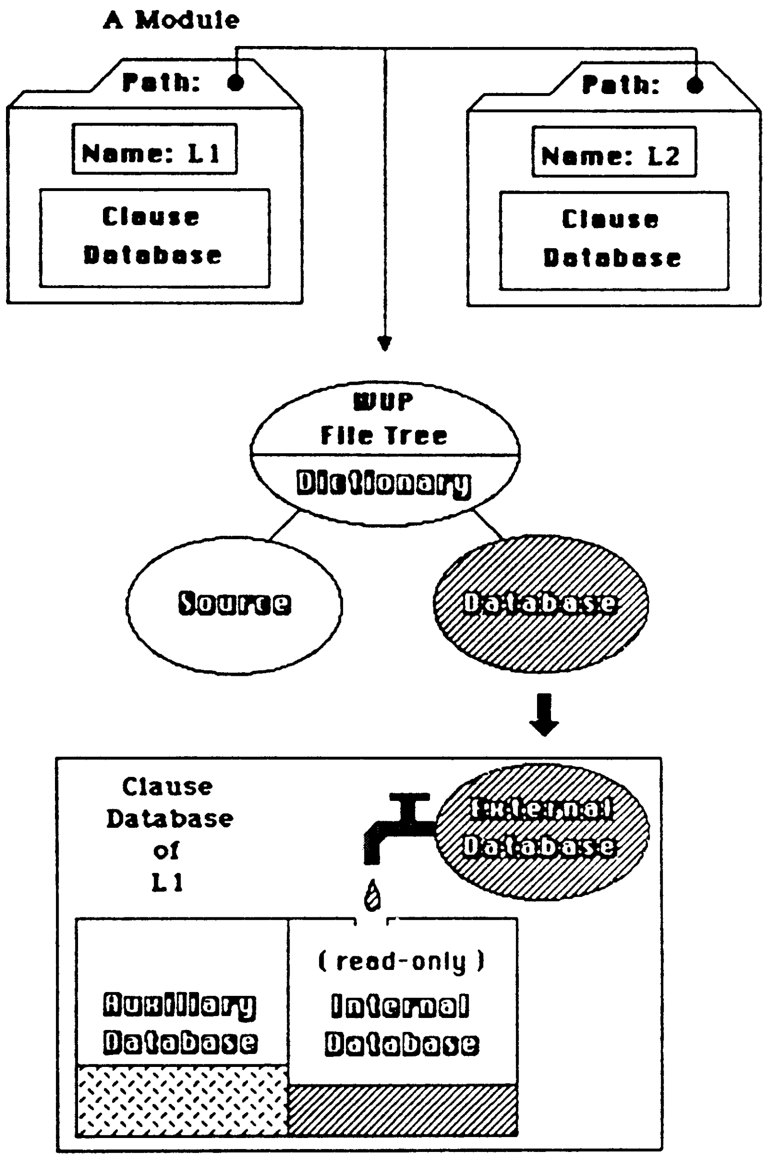


Figure 2.5 Definition of a Module

Facilities for supporting information-hiding has been incorporated in many modern programming languages (e.g. CLU, Ada). WUP provides a very simple mechanism for restricting the accessibility of certain predicates in a program module. A list of visible or accessible predicates can be specified in a special predicate source file (".export") inside the program module. Any predicates not specified in this file are strictly local to this program module, which means they can be accessed only by those predicates belonging to the same module. As a consequence, all clauses created at runtime in the auxiliary database are private to its program module.

The notion of program module in WUP provides a uniform view of Prolog programs and libraries. Large programs can be developed by combining many such program modules. Using a program module as a tool, one can also exploit the ideas of abstract data types and generic types. Furthermore, program modules can also be used to encapsulate domain-dependent data [Feu83]. Our concept of a "program module" and a "module" is analogous to a program and a process. A program module is a static collection of clauses, while a module is an execution instance of a program module.

In the following sections, we will describe the functionality and interaction of each WUP component. A more detailed description of module in Prolog from a user's viewpoint can be found in [Poo84] and [Che84].

2.3. Outline of System Structure

To increase modularity and extendability, WUP is organised into five major components. Each is constructed in such a way as to hide its internal structure as much as possible and to reduce the sharing of information to a minimum. The translation of a Prolog program to its interpretive code is done in three distinct stages: preprocessing, parsing and code generation. Searching, dynamic linking and loading, and proving are the phases of execution.

Preprocessing

The varied syntaxes of available Prolog systems makes it difficult to transport Prolog programs. In view of this difference, a preprocessor has been given the task of doing the syntax to syntax translation, interpreting operator declarations (cf. DEC-10 Prolog, CProlog, Waterloo Prolog) and expanding macros. With a fixed intermediate language, the expansion of operators simplifies the later task of the parser. Together with the macro facility, a user can define his own "syntactic-sugar" for readability. Furthermore, the discrepancy in the naming of built-in predicates from system to system can, hopefully, be resolved by means of macro facility (e.g. ESP) [Chi84].

Parsing

The elimination of operator declarations and macros by the preprocessor makes the lexical and syntactical analyses rather simple. In addition to performing a few semantic checks, the parser also ensures the matching of predicate names and their filenames. After an input clause is successfully parsed, its machine-independent intermediate code is generated and passed on to the code generator.

Code Generation

Our current code generator is specifically written for the translation of intermediate code to our interpretive pure code. The output of this code generator is a relocatable form of pure code which is then saved in the program database. An interesting aspect of this pure code is that the original input clause can be reproduced from it, in other words, the compilation process is reversible, and at the same time, it is compact enough for easy interpretation at runtime.

Clause Database

The program database structure is designed to facilitate fast retrieval and easy update on procedures. At the end of compilation, the relocatable pure code representation of a Prolog program source is saved into the corresponding program database. Later, during execution, a procedure is retrieved, dynamically loaded and link-edited into memory from the external program database. Since self-modifying programs are not permitted, any future references to the same procedure can be found in the internal database. Such a scheme makes code swapping (i.e. reloading) possible when main storage is exhausted.

Interpreter

This is the heart of the whole system and is based on the ABC algorithm [van81]. As in most sequential Prolog system, three stacks are maintained by this interpreter: a runtime (or control) stack, a copy (or global) stack and a trail (or reset) stack. Using these stacks, the interpreter generates, searches and maintains a proof tree. Whenever possible it also performs tail recursion optimisation [War80] and garbage collection on the copy stack [Bru82a]. All system predicates are trapped here, and control is passed to the corresponding system procedures.

Runtime Library

WUP has no directives or command language. Every input from the user specifies either a user or system predicate. All system in predicates written in the system implementation language reside in the runtime library. Since the rest of WUP has no prior knowledge about the existence of any pre-defined predicates, this permits us to tailor the set of system predicates to any particular application. This can be done without affecting the functionality of other components. It is one of the notable features in this new implementation of Prolog.

Chapter 3
Design Details

To make efficient use of a programming language, some basic tools are required: a text-editor, an interpreter or compiler, and a file system or database management system. With the development of larger programs, the separate compilation becomes increasingly important; it provides the basic tool for modular programming and reusable software. In recent years, most of these basic tools have been successfully integrated into a coherent programming environment (cf. Interlisp).

The design of WUP follows the same philosophy. Our aim is to provide a good program development system and to promote a programming paradigm based on modules in Prolog. For future extendibility, WUP is organised into several distinct but interdependent components. Although WUP is specifically tailored for the Unix operating system many of the system-dependent features are isolated for maximum adaptability and modifiability. This chapter will discuss how these components are designed and how they are interconnected to produce a usable programming environment.

3.1. Specification Notations

Here we describe our notation for use in our specification language. A program consists of two parts: data declarations and operation specifications. The declaration part uses equational notation to define the basic data types and their structures. For each operation, there is a type specification to indicate the types of parameters, and an action specification which uses Prolog with annotated variables to indicate its action conditions and sequences. The Prolog syntax used is the same as WUP's syntax which will be described in later section. The type of each parameter in every predicate is shown in the type specification position-wise. Every Prolog variable is either preceded by a "?" for an input-only variable or "!" for an output-only variable (cf. CSP) [Hoa78]. A predicate is not activated until all its input-only variables are instantiated. All its output-only variables, which must be free at the time of activation, will be bound at the end of its execution. For example, given the clause "p(?X) <- q(?X,!Y),r(?Y)", the predicate "p" and the subgoal "q" are not activated until "X" is bound, and "r" is not activated until "Y" is bound. We can view that "q" is the producer and "r" is the consumer of the channer variable "Y". Figure 3.1 is an example defining a stack as an abstract data type.

```
def Boolean = {true, false}
def Element = an element belongs to any domain type
def EmptyStack = []
def Stack = { [TopElement|RestElement], EmptyStack }

new-stack: Stack
new-stack( [] );

push: Element, Stack, Stack
push( ?E, ?S, [!E|!S] );

pop: Stack, Element, Stack
pop( [?E|?S], !E, !S );

is-empty: Stack
is-empty( [] );
```

Figure 3.1 An Example of Specification Notation

3.2. Frontend

The frontend of WUP consists of a *Preprocessor* and a *Parser* (see figure 3.2). The preprocessor is designed to make possible the translation from other Prolog syntax to WUP syntax, the intermediate language (IML). The translation of IML to WUP machine-independent Prolog intermediate code (PIC) is done by the parser. The *Code Generator* (see section 3.3.1) transforms PIC to pure code (PC) for WUP's interpreter. Each of these components has a well-defined interface, and provides enough flexibility for future modifications.

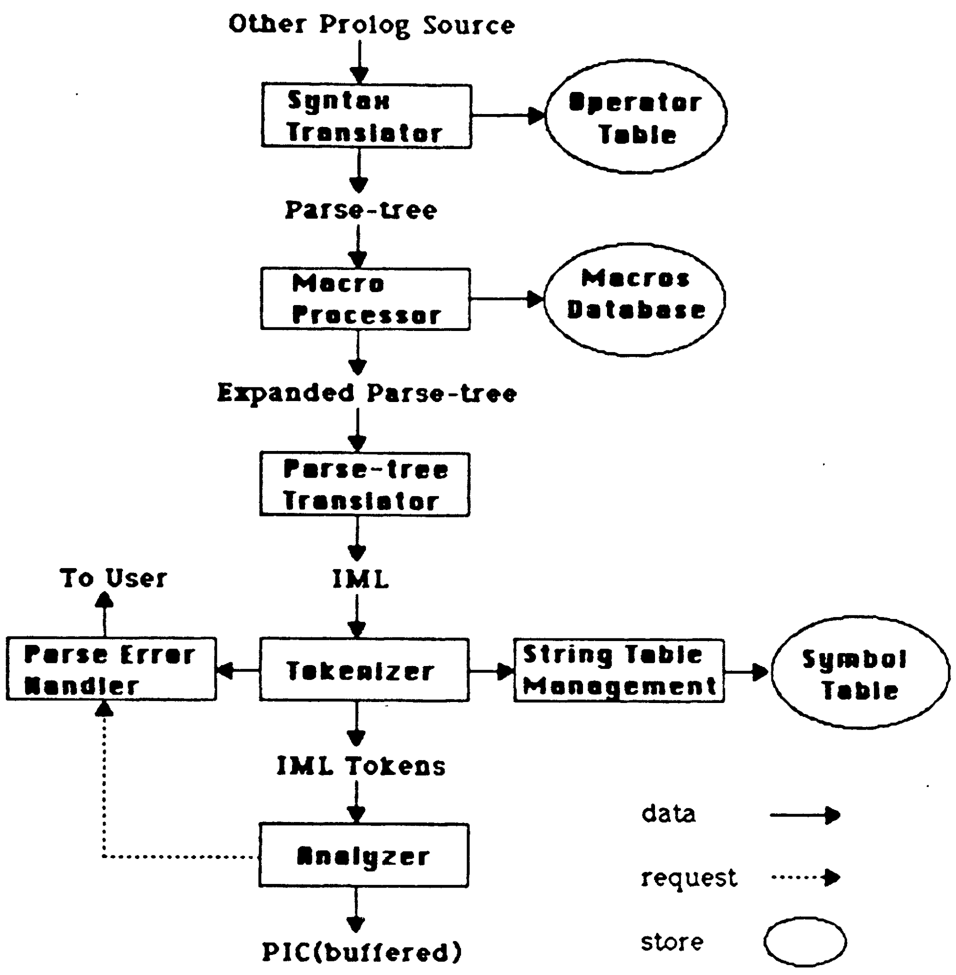


Figure 3.2 Frontend of WUP

3.2.1. Preprocessor

The preprocessor has three components, all of which hopefully will be implemented in Prolog itself †. The first component is the *Syntax Translator* where operator predicates are expanded. It collects all the operator predicates into a database, the *Operator Table*, and translates the input Prolog source program to a *Parse-tree* notation.

```
example:
        op( "<", left-to-right, 130 );
        op( "&", left-to-right, 140 );
from:
        X < Y & Y < Z
to:
        "&"( "<"(X,Y),"<"(Y,Z) )
```

The second component is the *Macro Processor*. Given any clause in parse-tree notation together with a macro database, it does the textual replacement of every subtree that has a macro entry.

```
example:
        macro( "<"(X,Y), lt(X,Y) );
        macro( "&"(X,Y), and(X,Y) );
from:
        "&"( "<"(X,Y), "<"(Y,Z) )
to:
        and( lt(X,Y), lt(Y,Z) )
```

The output from this component is the parse-tree representation of the input clause without operator predicates or macros. The third component is the transformation from parse-tree notation to IML notation, the *Parse-tree Transformer*. Since our IML (described below) has a prefix function notation, it is a simple task to translate the parse-tree notation to IML.

† At the time of writing, the preprocessor is not yet implemented.

3.2.2. Parser

Parsing can be split into three stages: *Source handling, Tokenising*, and *Analysing.* The source handler takes care of the filtering of comments and redundant white space; this can be done in the preprocessing stage. The tokeniser identifies all tokens recognisable by IML, and classifies them into categories for the use by the analyser. The analyser performs the syntactic analyses on each input clause. For each valid input clause, the analyser will generate its intermediate code for the code generator.

3.2.2.1. IML

The original syntax of IML was Lisp-like notation [Ng82]. Though IML is intended to be an intermediate language, we would like to use it for WUP system programming. To increase its readability, we modified IML to a prefix function form with a list notation as in DEC-10 Prolog (see Appendix A1). As list structure is heavily used in Prolog, we give it a distinct notation, and internally represent it differently from term for efficiency. In the following example, we give an example of the "append/3" in IML (note that "," and "<-" are not reserved).

```
append( [], List, List ) ;
append( [Head | Oldtail], List, [Head | Newtail] ) <-
        append( Oldtail, List, Newtail ) ;
```

Each token in IML is characterised by its type and value. Figure 3.3 is a list of tokens defined in IML, together with their corresponding reserved character representations. The underscore indicates a don't-care field. The detailed regular expression definitions of each token type can be found in

Appendix A2.

```
def IML-token = token( Token-type, Token-value )
def Token-type = { query, proc, atom, integer, char, comp-begin,
        comp-end, list-begin, list-end, restlist, null-list, variable,
        end-of-clause, end-of-input }
def Token-value = { integer-constant, char-constant, offset,
        string-descriptor }

examples:

        ? f( abc, [123 [] |X] ) ;

                token( query, _ );
                token( atom, f );
                token( comp-begin, _ );
                token( atom, abc );
                token( list-begin, _ );
                token( integer, 123 ) ;
                token( null-list, _ ) ;
                token( restlist, _ );
                token( variable, "Y" );
                token( list-end, _ );
                token( comp-end, _ );
                token( end-of-clause, _ );
```

Figure 3.3 IML Tokens Specification

3.2.2.2. PIC

Our intermediate code (PIC) is designed to be machine-independent as we would like to have different code generators for it in the future. The syntax of PIC is Algol-like with several pre-defined data types: atom, integer, single character, list and functor (see Appendix B). Variables in PIC are classified as: base, reference and void. A base variable is the first occurrence of a variable within a clause, a reference variable is the second or any subsequent occurences, and a void variable is a don't-care variable. List and functor are further classified as constant-type and variable-type. A constant-type structure (list or functor) is one not containing any variables; a variable-type structure does. (Note: the decision on choosing such a representation of structured objects and variables is affected by the representation scheme of "constructed" term at execution. (More detail on this later.) Each predicate or functor is identified by its name and arity. A predicate is a special kind of functor. Each clause in PIC also contains entries on the total number of base variables (those take up storage at runtime) and the number of subgoal calls in the body. Figure 3.4 shows the PIC form of the "append/3" program given earlier and figure 3.5 shows the specification of PIC.

Our initial version of PIC was specified as a form of abstract syntax tree [Aho77]. The present definition of PIC was chosen because it was very easy to introduce the semantic actions into the parser. On close examinination, one can see that PIC has no explicit information about the underlying machine architecture (e.g. word size).

3.2.2.3. Parser Specification

Now that we have defined the input and output of our parser, we can specify the required operations to support our design (see figure 3.6). Our top level driver of the parser is the *read-clause()* which coordinates the processing involved at each stage.

```
PROC 1 call 4 vars
          VAR_FUNC 3 args append
               VAR_LIST 2 terms 2 vars
                    BASE_VAR offset 1 Head
                    BASE_VAR offset 2 Oldtail
               END_LIST
               BASE_VAR offset 3 List
               VAR_LIST 2 terms 2 vars
                    REF_VAR offset 1 Head
                    BASE_VAR offset 4 Newtail
               END_LIST
          END_FUNC
     CALL
          VAR_FUNC 3 args append
               REF_VAR offset 2 Oldtail
               REF_VAR offset 3 List
               REF_VAR offset 4 Newtail
          END_FUNC
     END_CALL
END_PROC
```

Figure 3.4 PIC Representation of Append/3

3.3. Backend

The backend of WUP consists of the *Code Generator*, *Database Management System*, the *Intepreter* and the *Runtime Library.* The code generator translates the intermediate code PIC to the relocatable Pure Code (PC). The database management system maintains the storage or retrieval of clauses. The interpreter and the runtime library control and support Prolog execution. (see figure 3.7)

```
def PIC-objects = { PIC-constant, PIC-variable, PIC-functor,
            PIC-list, PIC-call, PIC-clause }

def PIC-constant = constant( Const-type, Value )
def Const-type = { atom, integer, char }

def PIC-variable = variable( Var-type, Name, Offset )
def Var-type = { ref, base, void }

def PIC-functor = functor( Func-type, Name, Arity, Arguments )
def Func-type = { var-func, const-func }

def PIC-list = list( List-type, Head, Tail )
def List-type = { var-list, const-list }

def PIC-call = call( PIC-skel )
def PIC-skel = { PIC-func, atom }

def PIC-clause = clause( Clause-type, NumCall, NumVar, Head-skel,
            [FirstCall|RestCalls] )
def Head-skel = { _, head( PIC-skel ) }
def Clause-type = { goal, proc }

examples:

    abc         constant( atom, "abc" )
    123         constant( integer, 123 )
    'c'         constant( char, 'c' )
    X           variable( base, "X", 1 )
    _           variable( void, _, _ )
    f(a,X)      functor( var, "f", 2,
                        constant( atom, "a" ),
                        variable( base, "X", 1 ) )
    ['a',Y]     list( var,
                        constant( char, 'a' ),
                        variable( ref, "Y", 2 ) )
    ? f(X) ;    clause( goal, 1, 1, _,
                        [call( functor( var, "f", 1,
                              variable( base, "X", 1 ))] ) )
```

Figure 3.5 PIC Specification

```
def In-stream = [First-char, ... , end-of-input]
def PIC-buf = pic( Clause-type, Name, Arity, PIC-clause )

read-clause : In-stream, PIC-buf, In-stream
read-clause( ?I1, !P, !I3 ) <-
        get-next-token( ?I1, !T, !I2 ),
        not-equal( ?T, end-of-input ),
        accept-clause( ?I2, ?T, !P, !I3 );

get-next-token : In-stream, IML-token, In-stream
get-next-token( ?I1, end-of-input, !I1 ) <-
        end-of-stream( ?I1 );
get-next-token( ?I1, !T, !I2 ) <-
        next-token( ?I1, !T, !I2 );

accept-clause : In-stream, IML-token, PIC-buf, In-stream
accept-clause( ?I1, token( query, _ ), !P, !I3 ) <-
        get-next-token( ?I1, !T, !I2 ),
        accept-query( ?I2, ?T, !P, !I3 );
accept-clause( ?I1, token( axiom, _ ), !P, !I3 ) <-
        get-next-token( ?I1, !T, !I2 ),
        accept-axiom( ?I2, ?T, !P, !I3 );
accept-clause( ?I1, _, _, !I2 ) <-
        report-error( ?I1, !I2 );
```

Figure 3.6 Parser Specification

3.3.1. Code Generator

Upon receiving a clause in PIC representation, the code generator produces the relocatable pure code (PC), which is then saved n a file, or link-edited for loading. This component also includes the link-editor, and is the only component in WUP that knows about the characteristics of PC. By abstracting clause as a data type, its integrity can be ensured, thus increasing the system's modularity.

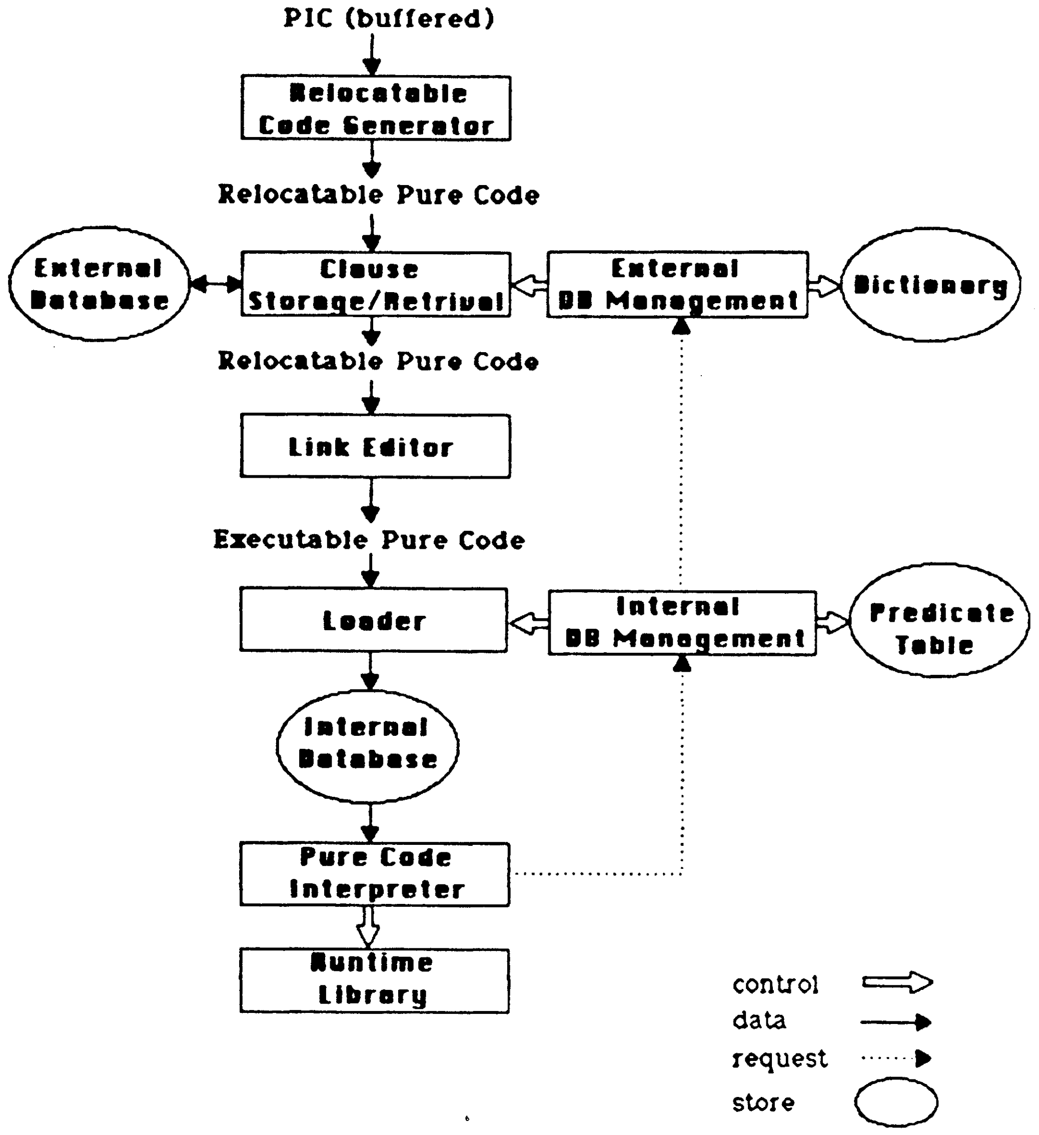


Figure 3.7 Backend of WUP

The PC is a form of threaded code and its data structure is a mixture of vectors and lists (see Appendix C). A functor is represented by a vector for fast accessing of its arguments. Lists and clauses are represented by different forms of list structure. A list is a list of cons cells (cf. Lisp); each has a head and a tail portion. A clause is a list with a head as its first element, and the list of subgoals as rest. As can be seen, there is a potential overhead for such a clause representation. This overhead is observed when one tries to implement meta-Prolog, Prolog in Prolog (which requires to convert a clause from a list into a functor and vice versa). It can only be eliminated if every structured object in the system is represented as a list of "cons" cell(s) (cf. Micro-Prolog) or as functors (cf. Waterloo Prolog). For instance a clause "p<-q" is represented as a term "<-(p,q)" in Waterloo Prolog and as a list "(p (q))" in Micro-Prolog. Each scheme has its disadvantage: the former uses more storage for the representation of a list constructor as a binary functor; the latter provides slower access to the arguments of a functor. At the expense of conversion overhead; our approach takes the advantages of both representations. In WUP, one cannot manipulate "p<-q" as a term "<-(p,q)" but rather as a list "[p,q]", and can only rely on the system to do the conversion.

The set of PC-objects is almost the same as the set of PIC-objects except that each PIC-object is packed into a single machine word (PC-word) or an array of PC-words. The fundamental unit of PC is a PC-word that has a tag field and a value field. The size of a PC-word depends entirely on the chosen machine implementation of this system. Figure 3.8 shows the "append/3" program in PC form.

At this point, we have specifed the definitions of our abstract machine code and the translation processes from Prolog source programs to machine code (see figure 3.9). The interpretation of this machine code will be described in a later section on the runtime system. The management of Prolog clauses by the

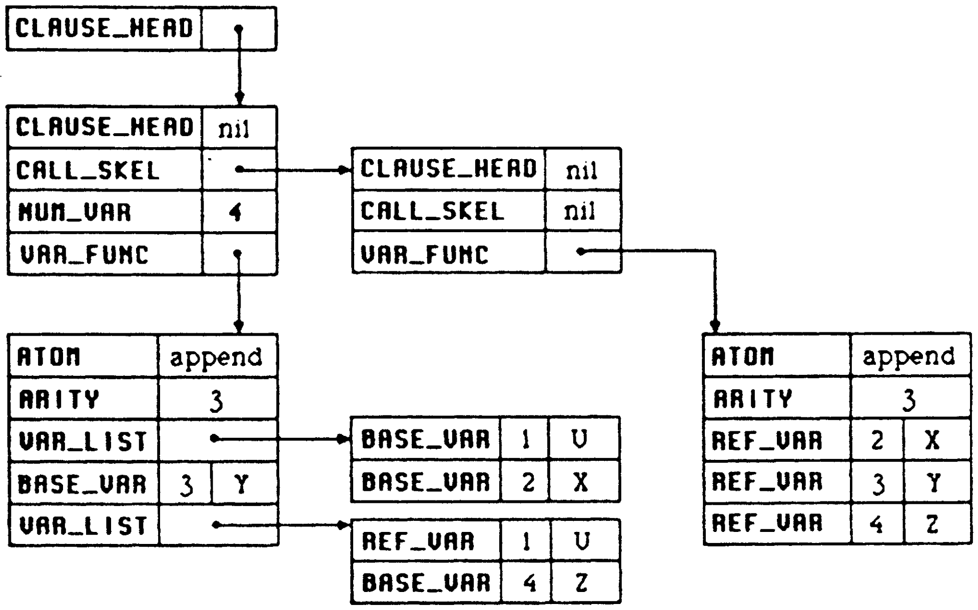


Figure 3.8 PC Representation of Append/3

database system will be the subject of discussion in the next section.

```
def PC-object = PC-word( Tag, Value )
def PC-type = { query, axiom }
def PC-clause = [Clause-head|Clause-body]
def PC-buf = pc( PC-type, Pred-name, Arity, PC-clause )

generate-code: PIC-buf, PC-buf
generate-code( pic(goal,?N,?A,?C1), pc(query,!N,!A,!C2) ) <-
      translate-body( ?C1, [], !C2 );
generate-code( pic(proc,?N,?A,?C1), pc(axiom,!N,!A,[!H|!B]) ) <-
      translate-head( ?C1, !C2 , !H ),
      translate-body( ?C2, [], !B );
```

Figure 3.9 Code Generator Specification

3.4. Database Management

Our database management (DBM) system is not a general purpose one; it is intended to be a secondary storage management system for Prolog clauses only. The program database structure resides on top of a tree-structure file system, and provides the foundation for implementing modules in Prolog. The fundamental ideas came from the Make system [Fel79] on Unix, and the file system of the Waterloo Port system [Mal83]. In Port every procedure is stored in a separate file and the procedure's name is the filename; the program as a whole is structured as a tree of files. Our objective is to provide a clause database management system for fast look-up of predicates and efficient maintenance of programs. Within the Unix file system, this sub-system is easily constructed and provides a well-organised interface between a user and his program modules.

In WUP, a procedure (a sequence of clauses with the same predicate name and arity) is defined as an abstract data type. Two well-defined operations on procedures are supported by this system: *clause-generator*() and *next-clause*() (see figure 3.10). Given any predicate name and its arity, the clause database manager will return the corresponding procedure upon calling *clause-generator*(). By successively invoking *next-clause*() on a given procedure, it will return the next clause in that procedure until there are none left. This is the only means by which the interpreter communicates with the clause database system.

```
def PredCall = call( Name, Arity )
def Procedure = [ First-PC-clause | Rest-PC-clauses ]

clause-generator: PredCall, Procedure
clause-generator( call(?N,?A), !Cg ) <-
        search-procedure( ?N, ?A, !Cg );

next-clause: Procedure, PC-clause, Procedure
next-clause( [?F|?R], !F, !R );
```

Figure 3.10 ABC Algorithm Support Specification

Clause DBM is split into three subsystems: module management, internal DBM and external DBM. The auxiliary DBM is merged into the internal DBM because there is only a minor difference on handling clauses in mainstore.

3.4.1. Module Management

The module management defines and controls the search strategy on predicates. There are two modes of searching for predicates: *Non-specific* and *Specific*. By default, given any predicate, it does a linear search along the search path starting from the caller's module; it searches downward and wraps around until it finds the required predicate. If the search returns to the caller's module, a warning is issued for a non-existent predicate. A predicate reference across modules is checked against its visibility; whether it is exported by the module in which it resides. This is the *Non-specific* mode search and it only looks up predicates in the main database.

A module can be viewed as a set of predicates performing some specific operations on some data (cf. abstract data types), or as a well-defined database which encapsulates some domain-dependent relations. In order to generalise this concept, all meta-predicates include an extra argument, the module name. For example, the "prove(X)" predicate is written as "prove(M X)", where M specifies the module in which X is to be proven. If M is unbound upon the execution of "prove", then the non-specific mode search is used and M will be bound to the name of the first module containing a definition for the predicate X. If M is bound, then only the module M will be searched. This is the *Specific* mode search, and both the main and auxiliary databases will be searched "without" visibility check. Hence, the user can have some amount of control over the search strategy being used for certain predicates. Given such facilities, one can also implement generic types in Prolog. Examples of the use of modules as generic data types can be found in [Poo84].

Coupling the different search strategies with modules facilities, the *clause-generator*() defined above should now be modified (see figure 3.11). Assume that the search path is <M1, M2, M3, ..., Mk>, the search sequence starting from M3 is <M3, M4, ..., Mk, M1, M2>. *Search-path*() returns a list of the current imported modules and *search-sequence*() generates the search sequence starting from Mi. All the other supporting functions will be described in the following sections.

```
def Mode = { specific, non-specific }
def Module = String

clause-generator: Mode, Module, PredCall, Module, Procedure
clause-generator( specific, ?M, call(?N,?A), !M, !P ) <-
        search-procedure( specific, ?M, ?N, ?A, !P );
clause-generator( non-specific, ?Mi, call(?N,?A), !Mo, !P ) <-
        search-path( !SP ),
        search-sequence( ?Mi, ?SP, !Mseq ),
        member( !Mo, ?Mseq ),
        search-procedure( non-specific, ?Mo, ?N, ?A, !P );

search-procedure: Mode, Module, PredName, Arity, Procedure
search-procedure( specific, ?M, ?N, ?A, !P ) <-
        search-main-db( ?M, ?N, ?A, !P );
search-procedure( specific, ?M, ?N, ?A, !P ) <-
        search-aux-db( ?M, ?N, ?A, !P );

search-procedure( non-specific, ?M, ?N, ?A, !P ) <-
        search-main-db( ?M, ?N, ?A, !P );

search-main-db: Module, PredName, Arity, Procedure
search-main-db( ?M, ?N, ?A, !P ) <-
        search-int-db( ?M, ?N, ?A, !P );
search-main-db( ?M, ?N, ?A, !P ) <-
        search-ext-db( ?M, ?N, ?A, !P ),
        assert-int-db( ?M, ?N, ?A, !P );
```

Figure 3.11 Module Management Specification

3.4.2. External Database Management

The external DBM takes care of all the object files that belong to a program module. It is the interface to the Unix file system and is the only part in WUP that is dependent on the existing file system. When we wish to move WUP onto a completely different file system, we can modify the interface between this component and the existing file system.

The program database of any program module is stored under a directory called ".db" (refer to figure 2.3). The dictionary is kept in a file ".dict" inside the program module, and is accessible to the user. Each entry in the dictionary specifies the name of a predicate and its absolute pathname in the Unix file system. Whenever a predicate file is added to (or removed from) the program source, a new entry is inserted into (or deleted from) the dictionary. This information is used to maintain the consistency of program source and its compiled program database.

For every predicate source file, there is exactly one predicate object directory with the same name inside the program database. If a predicate source file contains several predicates with the same name but different arities, then for each of the same arity there is an object file under that predicate object directory with its arity as the filename; this is called a *Predicate Object File*. With such a scheme, any predicate with a certain arity can be located in the program database. For example, when searching the predicate "p" with arity "2" in a module "M", we first compute the module path of module "M", say "path"; we can then test the existence of that predicate by accessing the predicate object file with pathname "path/p/2" through the file system primitives.

When compiling a program module, the external DBM traverses its program source file tree and compares the time stamp on each predicate source file against its corresponding predicate object directory in its program database. If there is a mismatch and the predicate source file is more recent, it is recompiled. If a predicate source file has no corresponding predicate object directory, the source file is compiled and a new directory is created. After it is recompiled, a new up-to-date dictionary is then saved. At runtime, the search for a procedure in a particular module from the external database requires one to check for the existence of the corresponding predicate object file in the program database (see figure 3.12).

```
search-ext-db: Module, PredName, Arity, Procedure
search-ext-db( ?M, ?N, ?A, !P ) <-
        module-path( ?M, !M' ),
        integer-to-atom( ?A, !As ),
        concatenate( ?M', ?N, !FP' ),
        concatenate( ?FP', ?As, !FP ),
        accessible( read, ?FP ),
        load-procedure( ?FP, !P );
```

Figure 3.12 External Database Specification

3.4.3. Internal and Auxiliary Database Management

This subsystem provides the necessary mechanisms for manipulating the in-memory clauses of modules. Whenever a procedure is requested by the interpreter through a call to *clause-generator()*, the internal database is first checked. If the procedure is not already loaded, the request is passed to the external DBM system. Using the pure code link-editor, the threaded pure code of any known procedure can be dynamically loaded into the internal database from the external database. Since the modification, insertion or deletion of clauses in the internal database is not allowed, we can release some of the currently unused clauses when storage is exhausted. This process of releasing clause storage can also be done by a user's explicit request, when he no longer needs them for further execution. Alternatively, it can be done automatically using a least-recently-used release algorithm†. In general, this is the case when a user maintains a large database, and only refers to a small portion of it.

For maximum modifiability, the structure of the internal database is also hidden within this sub-system. Different schemes for storing and retrieving clauses from a module can then be compared for efficiency.

The internal database consists of a *Predicate Table* and a list of procedures (see figure 3.13). The supporting function *search-int-pred-table*() is very similar to *locate-procedure*(). It needs to know the internal data structure of a module; the implementor can choose the appropriate representation. Given any module name, it should be able to find the corresponding predicate table and search the *Pred-entry* for any given predicate name. The organisation of the predicate

† This is not yet implemented in WUP.

```
def Proc-entry = proc( Arity, Procedure )
def Pred-entry = pred( PredName, [First-Proc-entry|Rest-Proc-entry] )
def Pred-table = [First-Pred-entry|Rest-Pred-entry]

search-int-db: Module, PredName, Arity, Procedure
search-int-db( ?M, ?N, ?A, !P ) <-
        search-int-pred-table( ?M, ?N, pred( !N, !PE ) ),
        locate-procedure( ?PE, ?A, !P );

locate-procedure: Proc-entry, Arity, Procedure
locate-procedure( [proc(?A,?P)|_], ?A, !P );
locate-procedure( [_|?R], ?A, !P ) <-
        locate-procedure( ?R, ?A, !P );
```

Figure 3.13 Internal Database Specification

table is also unspecified; it could be some form of hash-table in order to give fast access to the *Pred-entry*.

After a procedure is successfully loaded from the external database, it is then inserted into the internal database of the given module through *assert-int-db*(). Again the implementation of *assert-int-db*() is very much dependent on the chosen data structure of the predicate table.

The auxiliary DBM is almost identical to the internal DBM. The only difference is that the auxiliary DM will handle dynamic insertion and deletion of clauses at execution time. Every module has its own auxiliary database that can be accessed only by the predicates within the same module. New clauses can be added to any module's auxiliary database provided that it is not already defined somewhere in the module's main database. The *search-aux-db*() and *assert-aux-db*() are the same as their internal database counterparts. As long as the auxiliary database is used to store ground clauses only, the operation *retract-aux-db*() will not pose severe problems in the implementation. This can only be achieved by restricting the system primitives to only manipulate clauses in the auxiliary database.

3.4.4. Protection of Program Database

As mentioned in chapter 2, the visibility of certain predicates in any program module can be restricted. In WUP, an optional file ".export" can be used in any program module to list all predicates accessible to other module. Each entry in that file is in the form of a Prolog fact, "export(PredicateName)". This scheme can be extended to include the arity of a predicate or the name of a constant symbol.

To further support and encourage the use of program modules in Prolog, a simple locking mechanism is provided. If an optional (empty) file ".lock" exists inside a program module, then that program module is said to be locked. Any predicate that belongs to a locked module cannot be traced or listed during debugging (see section 3.6). A user can only see calls and the result of bindings before and after the execution of such predicate. With this feature, one can avoid tracing some predicates that have been fully debugged and tested, thus speeding up the process of program development. Since there is a unique interface between the DBM system and the interpreter, these proposed protection mechanisms can be incorporated without affecting the functionality of the interpreter.

3.5. Runtime System

The runtime system consists of two sub-systems: the *Interpreter* and the *Runtime Library*. The interpreter controls and drives the other WUP components at execution time and the runtime library supports all system built-in predicates. A user communicates with WUP through either his own predicates or system predicates; there are no directives or command language as opposed to CProlog. Every user query is eventually proven by the interpreter or executed by the runtime library routines. This has the advantage that the rest of WUP is insensitive to system predicates; hence, the runtime library can be tailored for particular applications without too much further effort.

The main interpreter loop is based on the ABC algorithm which has been modified to include several storage saving techniques [Bru82] such as popping deterministic computation and tail recursion optimisation. The runtime library is a collection of routines written in the implementation language C, and has a unique interface to the interpreter. Currently, all the library routines are compiled into WUP. Our design does not preclude the possibility of calling resident routines which may be written in some other languages (e.g. FORTRAN or Pascal), although this is not yet implemented in WUP.

In the following sections, we shall describe several improvements on the ABC algorithm, the unification algorithm used in this particular implementation, and the interface between the runtime library and the interpreter.

3.5.1. The ABC Algorithm

Using the two operations defined on the data type "procedure", the ABC algorithm generates and tests the AND-tree (i.e. proof tree) and the OR-tree (i.e. search tree) in response to a goal request. This algorithm defines the general structure of a stack node (or frame) to maintain all necessary information during execution. Two previous implementations of Waterloo Unix Prolog interpreter did not take advantage of the fact that a deterministic node requires less space. As much Prolog computation is deterministic, such a distinction can give a noticeable improvement on storage usage. As described by Maurice Bruynooghe [Bru82], given a goal and a sequence of candidate clauses for unification, a look-ahead on next possibly-matched clause is of great benefit; this can assist the early detection of a deterministic computation. In addition to saving the storage otherwise wasted for a non-deterministic node, it can help to determine the possibility of tail recursion optimisation. This idea is similar to the clause indexing technique mentioned by David Warren [War80].

To help to understand the basic terminology, figure 3.14 gives a description of the ABC algorithm. For any given clause, *new-environment*() will initialise a new binding environment, *head-of*() returns its clause head and *body-of*() returns its clause body -- a list of subgoals. The routine *unify*() takes a subgoal call and its environment and tries to unify it with a clause head and its new temporary environment (see next section). For any node, an empty procedure field (the clause generator for the call in the same node) implies that it is a deterministic node. Every new node generated in *son*() is pushed onto the node stack; the top node is popped one at a time to be the current node for checking its

```
def Environment = [Variable-bindings]
def Subgoals = [PredCall|RestPredCalls]
def Node = node( Subgoals, Procedure, Environment )
def NodeStack = [TopNode|RestNodes]

solve : Subgoals
solve( ?InitGoals ) <-
        new-environment( ?InitGoals, !InitEnv ),
        first-call-of( ?InitGoals, !C ),
        select( node(?C, [], ?InitEnv), [] );

select : Node, NodeStack
select( node([], _, _), [] ) <- succeed ;
select( node([], _, _), [node([_|?C], _, ?E)|?S] ) <-
        select( node(?C, _, ?E), ?S );
select( node([?C|?Cs], _, ?E), ?S ) <-
        clause-generator( ?C, !Cg ),
        son( ?Cg, node([?C|?Cs], _, ?E), ?S );

son : Procedure, Node, NodeStack
son( [], ?N, ?S ) <-
        backtrack( ?S );
son( [?Cl|?Cg], node([?C|?Cs], _, ?E), ?S ) <-
        head-of( ?Cl, !H ),
        new-environment( ?Cl, !T ),
        unify( ?C, ?E, ?H, ?T ),
        body-of( ?Cl, !C' ),
        select( node(?C', _, ?T), [node([?C|?Cs], ?Cg, ?E)|?S] ),
son( [_|?Cg], ?N, ?S ) <-
        son( ?Cg, ?N, ?S );

backtrack : NodeStack
backtrack( [] ) <- fail;
backtrack( [node(_, [], _)|?S] ) <-
        backtrack( ?S );
backtrack( [node(?C, [?Cl|?Cg], ?E)|?S] ) <-
        son( [?Cl|?Cg], node(?C, _, ?E), ?S );
```

Figure 3.14 ABC Algorithm Specification

"continuation" in *select*() and "alternatives" in *backtrack*(). The actions "succeed" and "fail" indicate the success and failure respectively of the proof of the original given goal. The following is a list of improvements that were included in this version of the ABC algorithm. The conditions which must be satisfied before applying these improvement techniques are then described.

[1] Deterministic and non-deterministic nodes are distinguished. In general, the size of a non-deterministic node is almost always twice the size of a deterministic node.

[2] Stack frames are popped at the end of a deterministic computation, deterministic call optimisation (DCO). All the stack frame storage generated during the execution of a clause can be released as long as there aren't any alternatives (or backtrack points) between the start and the end of such a computation. (Note: this includes the case when the alternatives are later removed by "cut".)

[3] tail recursion optimisation (TRO). Any deterministic recursive predicate call is unwound into iteration, which means that each recursive call is using the same stack frame storage. For deep recursions, this provides a big saving in terms of stack usage.

[4] clauses are indexed on the first argument of any predicate call. Given any predicate call, its first argument is used to select the first and next possibly-matched clauses. For a procedure containing a large set of clauses, this has the advantage of avoiding a fair amount of shallow-backtracking, and determines that such a predicate call is deterministic earlier, in order to apply the storage saving techniques described above.

In order to apply [1], every newly generated procedure through *clause-generator*() must have a single matched clause only. There are two cases where this could happen: either there is exactly one clause in that procedure, or through the use of [4] there is only one possibly-matched clause in that given procedure. For [2], the continuation (the unsolved subgoals in the current node) must be empty and there must be no backtrack point above (or more recent than) the current node. This applies to every node that is either deterministic originally, or later converted to deterministic by the "cut" predicate. For TRO [3], before unification, the current node for the current call must be deterministic or the current call must be the last one in the continuation of the current node, and the last backtrack point is above the current node. After unification, the newly created node for the matched clause is copied onto the current node and the new node is then released. Given the current call and its procedure (or clause generator), in order to apply [4], the current call must be a predicate call with arity greater than 0. To locate the next possibly-matched clause in the procedure given, we test whether the first argument of the clause head and the first argument of the current call are possibly-matched. Two terms are possibly-matched if: 1) either one is a free variable, 2) both are constant with the same type and same value, 3) both are list structures with their first elements possibly-matched, or 4) both are functors with same name and arity.

So far, we have only discussed how to save the storage on the runtime stack (or control stack). The trail stack can be collapsed to remove redundant entries. In particular, some of the variable binding records generated can be released at the end of TRO or the execution of the "cut" operation.

3.5.2. Unification

Our unification algorithm is the same as previous implementations, which were based on Robinson's algorithm [Rob65]. Figure 3.15 gives a description of a unification algorithm used in WUP. *Bind-to*() binds the term "T" with environment "E2" to the variable "V" in the environment "E1", and *identical*() checks whether two terms are the same term or have the same values. There is no "occurs" check in this algorithm. Most practical Prolog systems do not implement such a check in their unification processes, because it is too expensive. A Prolog system with more basic object types (e.g. WUP) can extend the given basic algorithm. In the next chapter, we shall give an example implementation of a more compact and efficient table-driven unification routine based on this one. A similar method was used in [Bal83].

```
def Term = term( TermType, Value )
def TermType = { const, var, func }
def TermList = [FirstTerm|RestTerms]
def Constant = term( const, Value )
def Variable = term( var, Value )
def Functor  = term( func, TermList )
def Env = binding environment

unify : Term, Env, Term, Env
unify( term(var, ?V1), ?E1, term(var, ?V2), ?E2 ) <-
        identical( ?V1, ?E1, ?V2, ?E2 );
unify( term(var, ?V), ?E1, ?T, ?E2 ) <-
        bind-to( ?V, ?E1, ?T, ?E2 );
unify( ?T, ?E1, term(var, ?V), ?E2 ) <-
        unify( term(var, ?V), ?E2, ?T, ?E1 );
unify( term(const,?V1), _, term(const,?V2), _ ) <-
        identical( ?V1, _, ?V2, _ );
unify( term(func,?A1), ?E1, term(func,?A2), ?E2 ) <-
        unify-arg( ?A1, ?E1, ?A2, ?E2 );

unify-arg : TermList, Env, TermList, Env
unify-arg( [], _, [], _ );
unify-arg( [?A1|?R1], ?E1, [?A2|?R2], ?E2 ) <-
        unify( ?A1, ?E1, ?A2, ?E2 ),
        unify-arg( ?R1, ?E1, ?R2, ?E2 );
```

Figure 3.15 Unification Algorithm Specification

3.5.3. Structure-sharing Versus Structure-copying

An interesting problem related to the runtime system is the construction of structured terms during unification. Bruynooghe [Bru82] and Mellish [Mel82] have discussed and compared the relative merits of using structure-sharing and structure-copying methods. Their results were inconclusive. They claimed that examples can be created to make each method highly inefficient compared to the other one.

Some recent attempts at defining virtual Prolog machines have proposed the use of structure-copying strategy [Bow83,Bal83]. They believed that structure-copying provides a better locality of reference in a virtual memory system, whereas structure-sharing has the potential danger of thrashing on a paging system. To provide the reader with a better understanding of this problem, we below describe the runtime structures of both methods for the goal call "? p(X);" and the fact "p(f(Y));".

Structure-sharing

1) Every newly constructed term is represented by a *Molecule*, which contains a pointer to the pure code of that term and a pointer to its binding environment (see figure 3.16). To construct a new term, it is only necessary to instantiate two pointers, hence structure-sharing allows rapid construction of terms.
2) When accessing the components of a structured term, and if such a term has many deeply nested sub-terms, a long chain of dereferencing is unavoidable. Therefore, in some cases, it is slower for accessing.
3) Sometimes, a newly constructed term may unify with a simple constant; in that case, a pointer could be wasted on a machine which cannot represent a molecule with a single machine word.

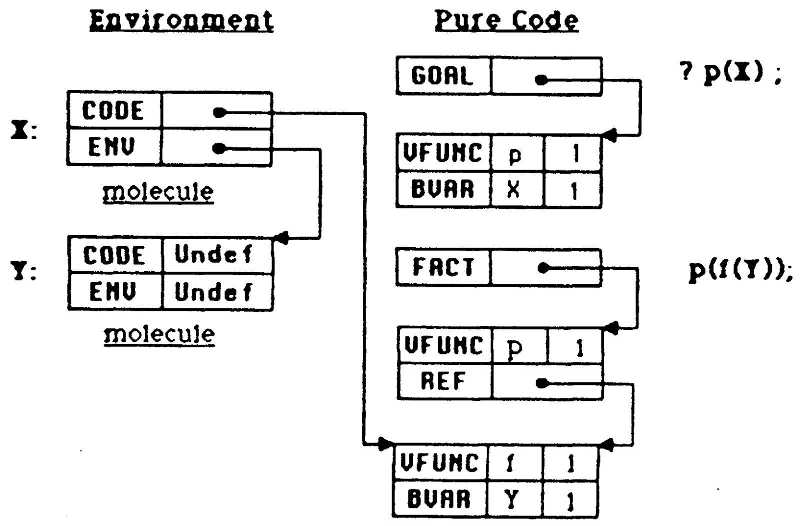


Figure 3.16 An Example of Structure-sharing

Structure-copying

1) To create a new structured term, a new copy must be made of the original pure code representation (see figure 3.17). Multiple occurences of the same variable must be linked together. For a large structured term, this could be very expensive in terms of space and time.
2) When accessing the components of a new structured term, it can be accessed directly from the new copy. Hence, it is faster for accessing.
3) When a new term unifies with a simple constant, there is no copying to be done. A simple assignment is all that is required. In fact, all "ground" sub-terms in any structured term can also be shared without copying. Therefore, for ground terms, there is no wasted time or space.

For the structure-sharing approach, in order to apply DCO and TRO mentioned earlier, some newly constructed terms in the "local" binding environment must be saved somewhere off the runtime stack to avoid "dangling pointers". (Recall that structure-sharing approach uses a molecule to represent a componud term containing variable(s). One of the pointers in a molecule is an environment pointer which refers to some binding environment. If some

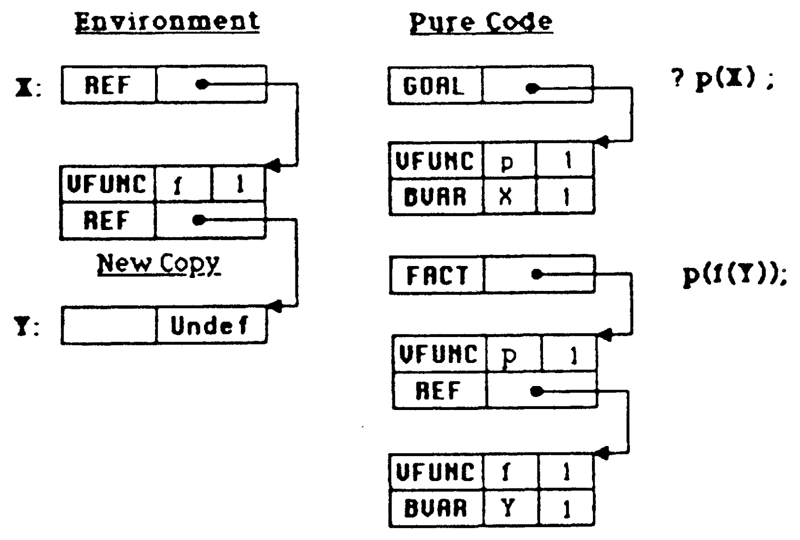


Figure 3.17 An Example of Structure-copying

environment is removed due to optimisation, then there is a danger of dangling references.) A new "global" stack is allocated for this purpose. However, since all terms to be contructed on the global stack cannot be identified at compile time, some are specified using the "mode" declarations (cf. DEC-10 Prolog) in order to reduce usage of the the global stack. In the case of the structure-copying approach, all structured terms are created on the "copy" stack at runtime. It is not necessary to identify the global or local status of every variable at compile time. As noted by Chris Mellish [Mel82], the distinction between "constructing" or "accessing" a term is best determined at runtime, which favours the structure-copying approach. In WUP, the structure-copying approach was selected for another reason; all previous implementations at Waterloo were based on the structure-sharing approach and we sought a better understanding of both representation schemes.

As mentioned earlier (see section 3.2.2.2), the reason for classifying structured objects into constant and variable types is to reduce the amount of copying at runtime; those that do not contain variables are "shared". The decision of choosing a runtime constructed term representation affects, to a certain extent, the design of intermediate and pure code. For example, if we choose structure-sharing approach, we need to identify the global or local status of each variable encountered at compile-time, and this piece of information must be stored somewhere in the intermedicate code and the pure code. Also we do not have to distinguish constant and variable structured terms. For a prolog system that does not intend to implement some of the storage optimisation techniques, the design of the frontend and the interpreter can be totally independent.

3.5.4. Interpreter and Runtime Library Interface

When designing the interface between the interpreter and the runtime library, two problems are commonly encountered. The first is the method of handling parameter-passing between Prolog code and non-Prolog code, and second is the method of identifying calls to non-Prolog code. It might seem that these two problems can be studied in isolation, but in some cases, their solutions are inter-related and must be considered together. To keep a system adaptable and modifiable, such an interface should have minimal interference with the other parts of the same system. Hence, one should be able to replace some of the runtime library routines with little knowledge of the design of the whole system. This is one of the major criteria set down at a very early design stage of WUP.

For the first problem, we use a standard protocol to extract and assign the parameters (as variable bindings) on the runtime stack through a common routine. In the current implementation, a dedicated area is used to store the parameters. When invoking a system predicate, the arguments in the corresponding environment are moved into that area; when returning from a system predicate, these arguments are then copied back into the environment. This can be done if we restrict parameter passing to non-structured terms only (e.g. integer, atom). For passing structured objects to a library routine, we can only use pointers and rely on that library routine to extract the necessary information. Obviously, such a library routine will need to know about the pure code representation of some structured object. By defining each structured object as an abstract data type, we can make the implementation of those library routines easier.

The second problem can be attacked in many different ways, but we are going to propose two such solutions and explain why one is better than the other. First, by initialising a table of all system predefined primitives, we could replace every call to such primitives by a (direct or indirect) pointer to the corresponding runtime routine at compile time. This will make the frontend (or some parts of the system) sensitive to some predefined primitives, and will complicate the matter of designing the parameter passing mechanism. In particular, each library routine will now have to handle (or extract) parameters. Furthermore, installing a new primitive could be difficult for a non-implementor.

The other possible solution is to create a Prolog clause instance for every system primitive, except that its clause body is some "pointer mechanism" to the corresponding runtime library routine. As long as these clauses are in existence before any calls being made, they can be trapped by the interpreter at execution time and passed on to a common system primitive handling routine which can set up the parameters and invoke the corresponding library routine. To install a new primitive amounts to creating a new clause instance and inserting a new entry in the system primitive handling routine.

The second solution is used in WUP because it fits well into our first solution and our concept of program module in Prolog. By pre-compiling the system predefined predicates into a module (the system library), they can be loaded upon their first reference. At interpretation time, the call to a system predicate is done in the normal way except, during execution of the body of such predicate, control is passed on to the system predicate handling routine. The pointer mechanism for each system predicate to its runtime routine is, in our design, the "case" label branching in the handling routine. This provides an interface between the interpreter and the runtime library having the advantage that the other components in WUP are insensitive to the changes of system primitives. For example, the built-in predicate "add(X,Y,Z) <- 15" is declared as a rule in the system library, when "add/3" is invoked during execution, the integer constant "15" is used as the "case" switch code in the primitive handling routine "CallSystem(Code)".

3.6. The Debugging Facility

Very often an interpreter is judged by the availability of program development tools. One of the most frequently used is a program debugger. Debugging a Prolog program is very different in nature from debugging a Pascal program. In Prolog, there is no name attached to each data object in the program, thus we cannot examine any kind of "variable" as in a Pascal program. Instead, we have to follow the control flow of a Prolog program and examine the variable bindings constructed by the unification of a goal and its matched clause. In writing a debugger for Prolog, two general approaches can be taken. One is a meta-level debugger written in Prolog and the other is an object-level debugger embedded in the runtime interpreter. A meta-level debugger executes and monitors a user program on behalf of the underlying interpreter on the system [Sha83a]. It has the advantage of being more flexible, portable and extendable. The object-level debugger is the Prolog interpreter itself but working in a special "trace" mode. It is simple and easy to implement in any Prolog interpreter.

We took the latter approach because it is very easy to introduce tracing facilities within the ABC algorithm. We follow the "box" model of Prolog execution described in [Byr80]. Every clause execution has four possible states: 1) "CALL" when it is first invoked, 2) "EXIT" when it succeeds on the given call, 3) "REDO" when backtracking occurs, and 4) "FAIL" when it fails on the given call. In trace mode, the predicate and its arguments with all variable bindings are printed for examination.

example:

```
[3,15] CALL: foo(_1) <alns|N> ?
       ...MORE...
[3,15] EXIT: foo(a) <alns|N> ?
[3,15] REDO: foo(_2) <alns|N> ?
[3,15] FAIL: foo(_2) <alns|N> ?
```

The tuple inside brackets, i.e. "[L,N]", is the unique identification of a node in the proof tree; L is tree level and N is the node number assigned in a pre-order traversal. The message "MORE" indicates the CALL just printed has alternatives. The trace modes are "<alns|N>", which stand for a(bove-level-spy), l(evel-at), n(ext-spy), s(ingle-step) and N(-skip). The meaning of each mode is:

a	show every spy point above the pre-selected tree level.
l	show every node at the pre-selected tree level.
n	show the next spy point.
s	show every step of execution.
N	when inside one of the above modes, an integer N has different meaning. For example, in mode "n", 5 means "skip 5 spy points", and in mode "s", 5 means "skip 5 single step of execution".

As in most other debuggers, breakpoints can be set on any predicate using a "spy" facility. A rather unusual tracing facility is the filtering of tracing steps at a certain level in the proof tree. The idea is to cut the proof tree in two at a particular level, and to display the tracing steps on or above that level only. When one is certain that a program error occurs above a particular tree level, this certainly helps to speed up the process of debugging. Together with the locking mechanism in modules, WUP provides a fairly simple and useful debugger.

3.7. Built-in Predicates

Following the same direction as in the second version of Waterloo Unix Prolog [Ng82], our first attempt is to define a minimal set of system predicates. Since there is no standard set of system predicates that should be included in any Prolog system, this approach seems reasonable. However, defining a minimal set of system predicates is not easy. In particular, to support a whole new programming environment in Prolog we must define a set of primitives which is sufficient to construct such an environment in Prolog itself. Only experience will tell whether these primitives can be efficiently implemented in Prolog or in system implementation language.

> The task of designing interactive programming systems is hard because there is no way of avoiding complexity in such a system.... The only applicable research method is to accumulate experience by implementating a system, synthesize the experience, think for a while and start-over [San78].

In WUP, all built-in predicates written in the implementation language are pre-compiled and stored in the system library. All other predicates that can be written in Prolog are stored in a standard library. They are referred to as the module "sys" and "std" respectively. All predicates defined in these libraries can be redefined by the user. WUP does not verify that predicates have been defined by the system or other program modules. There are advantages (A1-A2) and disadvantages (D1-D2) to this approach:

A1) A user may want to distribute a relation into several program modules because they are domain dependent; depending on the input for a proof on such relation, the user can switch from module to module efficiently. He cannot do so if WUP checks every predicate to see whether it is already defined in another module.

A2) Sometimes a certain predicate cannot achieve a user's intent. Without resorting to defining the same predicate with strange names, a user can redefine it in some other module for specific purposes.

D1) Unnoticed by the user, the default searching path may uncover a predicate that has been defined elsewhere, and which might give some mysterious result or failure.

D2) The user must know every visible predicate in the search path, or he must explicitly state, for any goal, the module in which the goal should be attempted.

Our objective is to make every built-in predicate look like any user predicate, so that we can maintain the uniform view of program module given to all Prolog programs in WUP. The arguments mentioned above are only observations; they are not strong enough to make better design decisions.

For details on all built-in predicates defined in WUP, consult the implementation manual [Che84]. The basic environment support, such as editing, compiling, program listing and checking, and module handling, can be found in the tutorial and reference manual [van84].

Chapter 4
Implementation Details

WUP is written entirely in the programming language C and runs on the Unix operating system. C was chosen because of its good development tools and easy interface with the Unix† programming environment. The discussion in this chapter will be informal. No prior knowledge about Unix or C is assumed. The terminology used here will refer to that defined in the last two chapters.

This chapter is about the implementation details of WUP, most further details are available in the implementation manual [Che84]. Our discussion here will be mainly concerned with the runtime system of WUP: the interpreter and its supporting library. Other WUP components, such as the parser, the code-generator, the symbol-table management, the Unix-system interface, the error-handling and the database management system (DBMS), will not be described in this document. Most of these components, except the DBMS, are fairly standard components as in many other language processors. Our DBMS is a very simple clause database for Prolog and its implementation is rather straight-forward with the specifications given in the previous chapter.

In this chapter we begin with a description of WUP's storage organisation. We then introduce the different algorithms used in execution, unification and structure-copying. We explain the implementation of some of the more important built-in predicates and show how the trace facility can be

† University of California, Berkeley, Software Distribution 4.2 version 7.

incorporated into the ABC algorithm.

4.1. Runtime System

The runtime system of WUP consists of the interpreter and the runtime library. The interpreter is based on a slightly modified version of the ABC algorithm, which implements the "prove" built-in predicate. The runtime library is a collection of C routines supporting all other built-in predicates. The memory organisation of our interpreter is similar to other Prolog implementations. Three stacks (runtime, copy and trail) are used to maintain the state of computation. The unification algorithm is almost the same as previous WUP implementations except that it is table driven. As mentioned in chapter 3, WUP uses the structure-copying strategy for constructing terms at execution time; this strategy is tightly coupled with the unification algorithm. There are many built-in predicates defined in WUP. Some are for supporting WUP as a programming environment and others for making Prolog a better programming tool. They are too numerous for discussion and only a few will be described here. In the next few sections, we will describe all these aspects of our runtime system in detail and explain the storage optimisation problems that were solved during our implementation.

4.2. Storage Organisation

In WUP, three stacks are used to maintain the state of execution: the runtime (or control) stack, the copy stack and the trail stack. The runtime stack stores the control (or navigational) information and the binding environments of variables. The copy stack saves the terms constructed during execution. The trail stack records pointers to variable bindings which are instantiated during unification and are undone (reset to "undefined" values) upon backtracking.

Runtime Stack

Two types of object are stored on the runtime stack: stack frame and environment. A stack frame corresponds to an activation record as in the implementations of block-structured programming languages; an environment corresponds to the storage allocated for parameter(s) and local variable(s) as in an Algol-like procedure. A stack frame is created for every predicate call and its matched clause-head pair. The environment for the matched clause, with size equal to its total number of unique variables, is allocated on top of the stack frame (see figure 4.1). A stack frame together with its binding environment (possibly empty) is called a "stack node" (see section 4.3 for further details).

Since many Prolog computations are deterministic, the distinction between a deterministic (DET) and a non-deterministic (NDET) frame could (depending on the program) result in a substantial saving in storage. A DET frame has three fields; a NDET frame has seven (see figure 4.2). The meaning of each field is defined as follows:

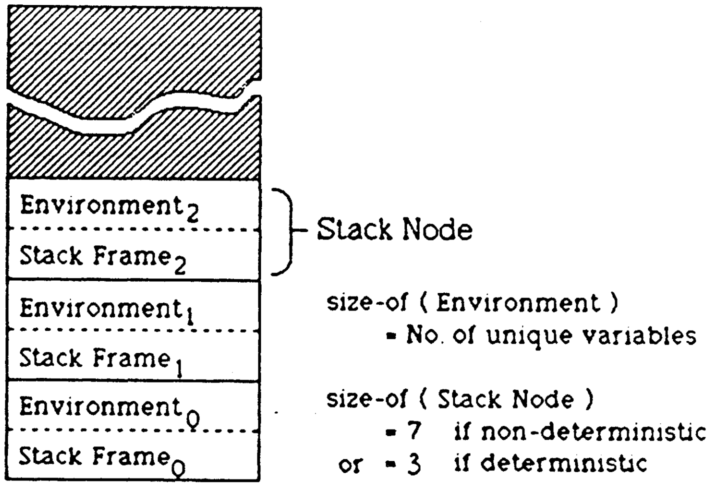


Figure 4.1 Runtime Stack

FATHER	a stack frame pointer to the parent frame of the current node. This forms the chain of all parent nodes belonging to the current node and thus maintains the structure of the proof tree.
CALL	a pure code pointer to a subgoal call of the clause causing the creation of the current node. It provides access to all the remaining subgoal call(s) within the same clause (this is where the computation will continue after successful completion of the current call).
MODULE	a pointer to the module containing the matched clause of the current call. This is necessary for implementing multiple modules in Prolog; it helps find predicates. (Note: The module from which the current call originates might not be the same as the module for its matched clause. In this case, the module from which the current call originates is not in the current stack node, but in its parent stack node.)
CL-GEN	a pointer to pure code for the procedure containing the remaining candidate clause(s) for unification with the current call. When the current node is selected for backtracking, this provides the remaining alternative clause(s) for the current call, which are then retried.

BACK	a stack frame pointer to the most recent backtrack node (or choice point). This forms a chain of all backtrack nodes of the current proof tree. Upon failure this speeds up locating the last backtrack node; the segment of runtime stack above it is then popped.
RESET	a pointer to the trail stack that records the top of the trail stack at the time of creation of the current node. On backtracking, it indicates the segment of trail stack to be popped; each variable binding recorded in this segment is reset.
COPY	a pointer to the copy stack which records the top of the copy stack at the time of creation of the current node. Upon backtracking, it is used for indicating the segment of copy stack to be popped. During unification, variable bindings that occur above this pointer on the copy stack are not recorded (see section 4.3 for further details).

<u>Deterministic</u>

CALL	→ PureCode
FATHER	→ Stack Frame
MODULE	→ Module

<u>Non-deterministic</u>

CALL	→ PureCode
FATHER	→ Stack Frame
MODULE	→ Module
CL-GEN	→ PureCode
BACK	→ Stack Frame
RESET	→ Trail Stack
COPY	→ Copy Stack

Figure 4.2 Stack Frame

The Copy stack

The copy stack has a function similar to that of the heap in Pascal. Its main purpose is to save the newly constructed terms at runtime. It is like a heap because all objects are created dynamically during execution; during their life time some may become inaccessible and must be garbage-collected. It is not exactly a heap because some objects can be released in a "last-in first-out" fashion, so that garbage collection is not the only means of reclaiming storage. A new object is created on the copy stack at runtime under the following circumstances: (1) during unification, whenever a free (unbound) variable binds to a structured object (functor or list) containing variable(s), a new copy of that structured object is made on the copy stack and a pointer to it is assigned to that free variable; (2) whenever there is a structural conversion, e.g. functor to list, the final object (a list in this case) is temporarily saved on the copy stack; (3) implementations of some meta predicates (e.g. prove, assert) require internal objects to be created in order to proceed: these internal objects are also allocated on the copy stack.

Objects are released, and their storage on the copy stack is reclaimed, under the following circumstances: (1) upon failure of a predicate call, its stack node and "copy stack segment" -- all the storage used on the copy stack during its execution -- are popped; (2) when the copy stack overflows, some of the inaccessible objects are then garbage-collected (this is not yet implemented at the time of this writing).

In the following example program, the storage allocated on the copy stack during the execution of "process" becomes inaccessible upon the next recursive call; hence, this must be garbage-collected. (Note: this problem also exists in structure-sharing approach using a global stack.)

```
cycle( S0 ) <-
      process( S0, S1 ),
      cycle( S1 ) ;
```

The Trail stack

The trail stack stores pointers to the environment or copy stack. During unification, the addresses of newly bound variables are recorded on the trail stack. Upon backtracking, these addresses are used to reset the variable bindings to "undefined". Because of Prolog's left-to-right depth-first execution order and its stack-based storage organisation, not every variable binding occurring during unification is recorded. As described earlier, upon backtracking, all stack nodes above and including the most recent backtrack node and their corresponding copy stack segments are popped. Hence, only those variable bindings in the environments (copy stack segments) below the most recent backtrack node's environment (copy stack segment) are recorded. (Note: The "cut" operations in Prolog can freeze the variable bindings in some stack nodes; thus their binding records on the trail stack can be thrown away as part of the storage optimisation.)

4.3. Memory Management

In the previous chapter we mentioned briefly some of the storage saving techniques implemented in WUP. Before we embark on our discussion of the improved ABC algorithm, we first introduce some of the basic concepts in the memory management of Prolog. We begin with a propositional and deterministic case to show the internal structure of the runtime stack. Next we show the environments and their variable bindings for a non-deterministic computation, and how the trail stack is used to record and undo the variable bindings. We then illustrate the structure-copying method for creating new data structure at execution time. Finally, we demonstrate, by an example program, how these concepts are put into use in the storage saving techniques. (Note: All stack addresses in these examples grow higher as the stack grows downward. This is important when stack addresses are compared to determine which way stack pointers should point.)

Example 1: Propositional and Deterministic

In figure 4.3, we show a propositional deterministic program with its resulting internal structure of the runtime stack after execution. Initially, we assume that there is an implicit call '?' generated by the system which will match any user queries and cause the creation of the first stack node N1. The FATHER of N1 is obviously nil since N1 is the root of the proof tree. The CALL of N1 is the implicit query '?'. (Reminder: A stack node is created for every call and its matched clause-head pair.) The node N2 is created for the first call 'p' in the original query and the clause-head of rule C1. The CALL of N2 points to the call 'p' and the right-hand brother of 'p', i.e. 's', and its FATHER points to the node which initiates the call, i.e. N1. Note that we do not store the head of the matched clause explicitly in any node; its body, if any, is given by the CALL field of its left-most child node. For instance, the body of the matched clause of '?' in node N1 are listed in the CALL of node N2, and the body of the matched clause of 'p' in node N2 are listed in the CALL of node N3, etc.

Selecting an unsolved subgoal can be divided into two steps: (1) Forward Selection (FS) - the selection of first subgoal in any matched clause; (2) Backward Selection (BS) - the selection of next subgoal in any matched clause. The FS step can be done without examining the stack nodes. Given any matched clause, we just pick off the first subgoal in its body as the next unsolved subgoal. If FS step fails, which implies the body of the matched clause is empty, then we have to follow the FATHER chain. Starting from the current node, if the next call in CALL is not nil, then the BS step succeeds and we have found an unsolved subgoal; otherwise, we repeat the same step for the parent of the current node. When the BS step has exhausted the remaining subgoals, the proof of the original input query succeeds.

By a matched clause "exit" we mean that the last call in that clause body has completed successfully and the control is passed back to its caller. If a matched clause exits deterministically, i.e. itself and every immediate subgoal call invoked by this clause are deterministic, then the storage occupied by these nodes can be released. We observe that popping DET nodes can be performed only when we are in the BS step. During every BS step, if a DET node is found

```
Goal:  ? p , s ;
C1:    p <- q , r ;
C2:    q <- s ;
C3:    r ;
C4:    s ;
```

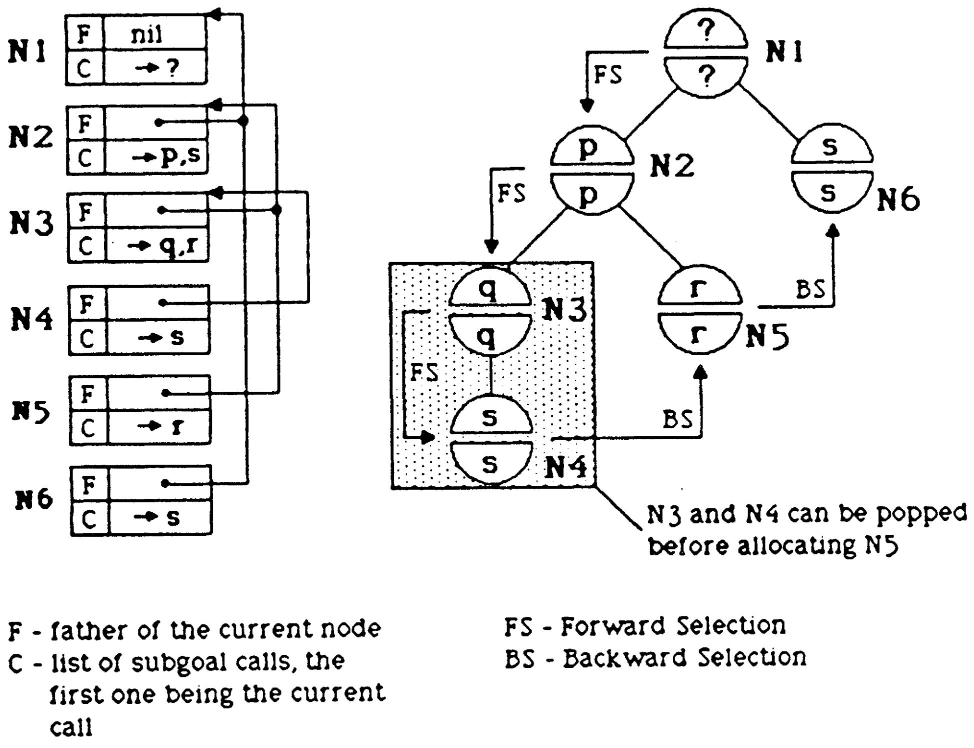

N1
N2
N3
N4
N5
N6
F
C
nil
→?
→p,s
→q,r
→s
→r
→s
FS
BS
?
p
q
r
s
N3 and N4 can be popped before allocating N5
F - father of the current node
C - list of subgoal calls, the first one being the current call
FS - Forward Selection
BS - Backward Selection

Figure 4.3 An Example of Propositional and Deterministic

to be "above" the most recent backtrack node, then it is popped. For instance, where the clause C2 (with immediate subgoals in node N4) exits successfully, the nodes N3 and N4 can be removed just before the creation of node N5. As we can see, it is only safe to do so if we can guarantee that there are no pointers oriented from either N1 or N2 to N3 and N4. Therefore, all pointers within the runtime stack are always oriented downward, from top to bottom in order to avoid the danger of dangling pointers. This is one of the major assumptions in our subsequent discussions on storage optimisations.

Example 2: Backtracking with Enivronment

In figure 4.4, we show a non-deterministic program with variable bindings. In order to illustrate the basic ideas, we will not perform any of the possible storage optimisations. All variables names are uniquely defined and are referred to globally for convenience. Given any subgoal call, we store its binding environment in its parent node. For example, in figure 4.4a, the binding environment for the call 's(U)' in node N3 is in node N2, and for the call 'r(Y)' in node N4 is in node N1.

In the node N1, an environment of size 2 (X and Y) is allocated for the original input query (see figure 4.4a). Node N2 is created with an environment of size 1 (U) for the execution of the subgoal call 'q(X)' and its matched clause C1; its RESET field points to the beginning of the trail stack and the CL-GEN points to the next alternative clause C2. After unification, the free variables U (in N2) and X (in N1) are unified, and a pointer is assigned from U to X. The execution of the call 's(U)' and its matched clause C5 (node N3) causes U to unify with the constant 'a' which is then assigned to X after dereferencing, and X is recorded on the trail stack because it is below the most recent backtrack node N2. Similarly, after the execution of the second subgoal 'r(Y)' (node N4) in the input query, Y unifies with the constant 'b' and is also recorded on the trail stack. Node N5 is created for the subgoal 't(X,Y)' and the matched clause C3; Z first unifies with X and gets the constant 'a', but then fails to unify with Y; the

```
Goal:  ? q(X), r(Y), t(X,Y) ;
C1:    q(U) <- s(U) ;
C2:    q(V) <- r(V) ;
C3:    t(Z,Z) ;
C4:    r(b) ;
C5:    s(a) ;
```

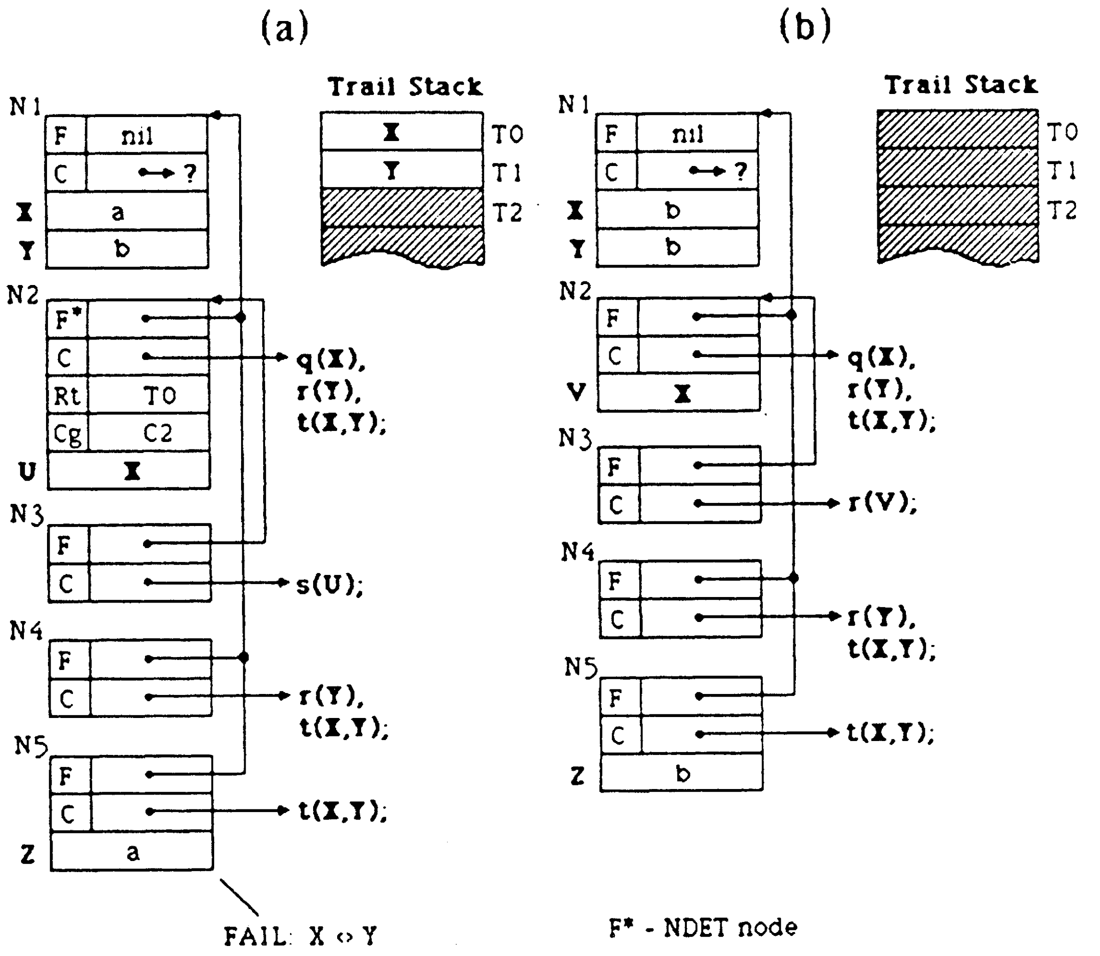


Figure 4.4 An Example of Backtracking with Environment

call 't(X,Y)' fails and causes backtracking. (N.B. Z is not recorded on the trail stack because it is above the last backtrack node N2.)

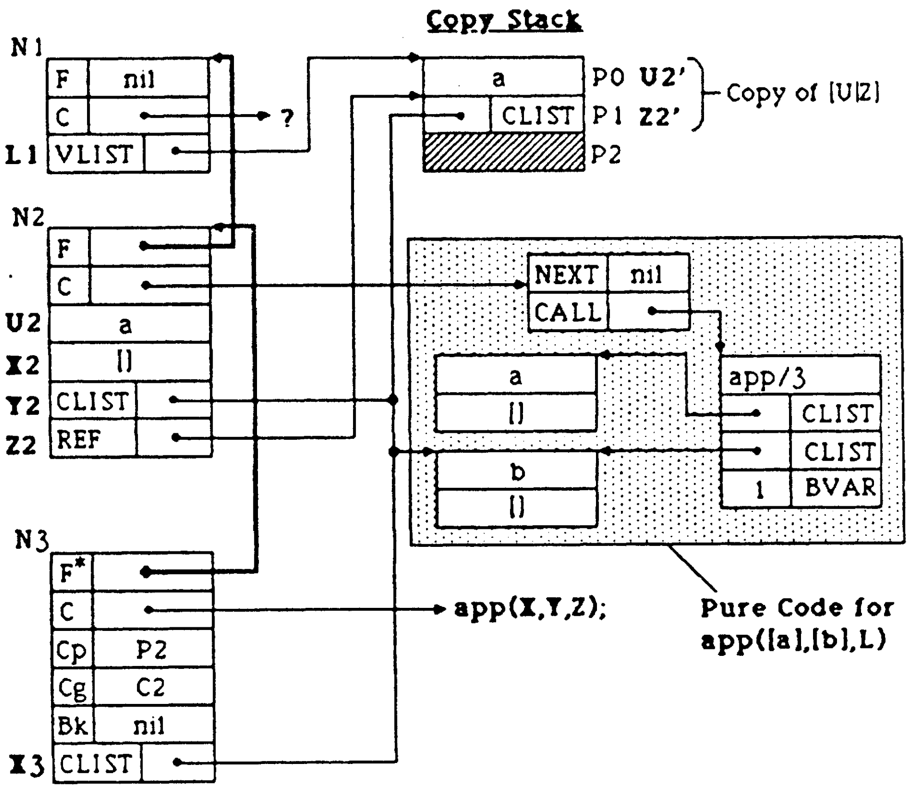
Upon backtracking, the nodes N5, N4, N3 and N2 are popped; every variable recorded from the top of the trail stack down to the pointer RESET in node N2 is reset to "undefined", which in this case is X and Y. In figure 4.4b, we show the situation after backtracking and the final execution. Notice that the node N2 is now DET because the clause-generator for 'q(X)' is empty, and nothing is recorded on the trail stack. The node N3 for the call 'r(V)' this time causes the variable X (in N1) to unify with the constant 'b' and the node N4 for the call 'r(Y)' causes the variable Y (in N1) to unify with the constant 'b'. Finally, the execution of the node N5 succeeds with Z equal to X and to Y, which is the constant 'b'.

In this sample program, we observe: (1) variable bindings occurring above the last backtrack node are never recorded on the trail stack; (2) simple constants are assigned to the free variables directly during unification; no pointer is needed and nothing is allocated or copied (3) when two free variables are unified in the runtime stack, the higher one always points to the lower one.

Example 3: Constructing New Terms

In figure 4.5, we show, with the "append/3" program, how new terms are constructed on the copy stack during execution. For simplicity, we omit the details of backtracking and show only the successful nodes in the runtime stack and the content of the copy stack.

```
Goal:  ? app( [a], [b], L ) ;
C1:      app( [], X, X ) ;
C2:      app( [U|X], Y, [U|Z] ) <-
             app( X, Y, Z ) ;
```



Note: Tag fields are reversed in some entries for convenience.

Figure 4.5 An Example of Constructing New Terms

Node N2 is created for the first and only subgoal call in the input query and the clause C2, which is DET with an environment of size 4. The variables U2 and X2 unify with the list '[a]' and get the constants 'a' and '[]' respectively; Y2 gets the list '[b]' (Note: variables are subscripted with their node numbers). When L1 in N1 unifies with a list structure '[U|Z]' with variables, it is assigned a pointer to a copy of '[U|Z]' on the copy stack; the new copy has U2' being replaced by the constant 'a', and Z2' being linked together with the corresponding entry in its environment. In order to avoid dangling pointers to the environment due to storage optimisation†, when a free variable in the environment unifies with a free variable in the copy stack, a pointer is always assigned from the environment to the copy stack. Hence, the variable Z2 in node N2 is assigned a pointer to the corresponding variable, Z2', in the new copy on the copy stack.

For the execution of the subgoal in clause C2, a NDET node N3 with an environment of size 1 is allocated for the matched clause C1; the COPY field is set to the top of the copy stack, P2. The variable X3 first unifies with Y2 and gets the list '[b]', and then it unifies with Z2 (and hence Z2'). The final value of L1 in node N1 is the list '[a,b]'.

In this example, we see that the decision to construct new terms on the copy stack is determined at execution whenever a free variable unifies with a structure containing variable(s). If the variable L in the original input query is replaced by a list structure '[a,b]', then no copying is necessary during execution.

† Recall that storage optimisation will remove portions of the runtime stack, therefore the potential lifetime of environments is shorter than corresponding copy stack entries.

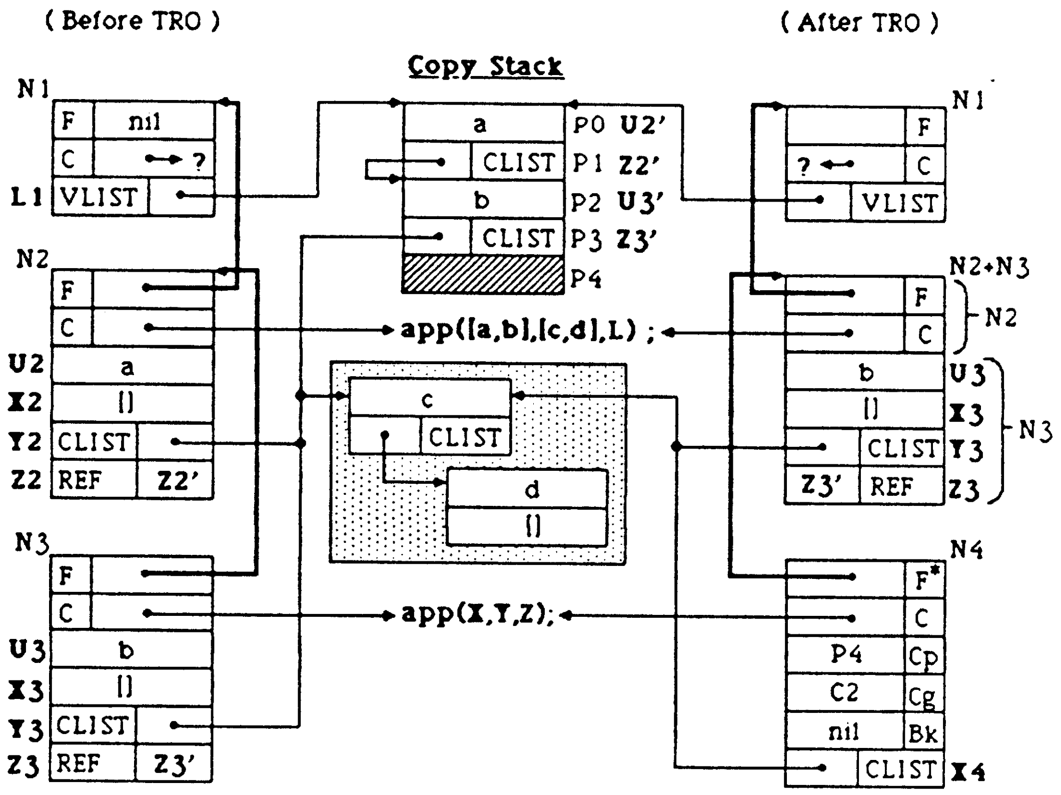
Example 4: Tail Recursion Optimisation

In figure 4.6, we show the application of tail recursion optimisation (TRO). The basic idea is to detect situations in which we can turn recursion into iteration. In Prolog, the only mechanism for performing repetitive computation is recursion. For a deep recursive call, the storage usage can be substantial. If we can make a recursive call re-use the same storage (a stack node), then we can save a lot of space.

In applying TRO, we must guarantee that whatever is left in the previous stack node is not relevant to the new one overwriting it. The conditions under which we can apply TRO are: (1) the call being invoked must be the last call in the clause to which it belongs; (2) in the same clause, between this call and the first call inclusive, there must not be any backtrack points; (3) the call is recursive (N.B. this is not a condition -- more on this later). For our example in figure 4.6, the call "append(X,Y,Z)" in clause C2 is the last call and recursive. Only at runtime can we tell if this call is deterministic or not. So long as the first argument X is not the empty list, this call will remain deterministic.

Following the first two steps of execution in our example, nodes N2 and N3 are created for the first subgoal in the query and the first recursive call respectively. Both are determinate and the variable bindings in N2 are on longer relevant for the future execution of N3. (However, there might be free variables in N3 which unify with the free variables in N2, and pointers are normally assigned from N3 to N2. Hence, there could be a danger of dangling pointers if we discard or overwrite N2. To avoid this problem, the solution here is to swap

```
Goal:  ? app( [a,b],[c,d],L ) ;
C1:      app( [], X, X ) ;
C2:      app( [U|X], Y, [U|Z] ) <-
              app( X, Y, Z ) ;
```



Note: Tag fields are reversed in some entries for convenience.

Figure 4.6 An Example of Tail Recursion Optimisation

the two environments -- but not the stack frames -- before unification, once we detect the possibility of applying TRO.) [Bru82] We observe that the CALL field in N3 is useless because it is the last call, and the FATHER field can be ignored since it is going to share the necessary information with its parent node N2;

therefore, N3's stack frame can be discarded and its environment can be placed over the environment of N2. As a result, N2 becomes a "hybrid" node -- its stack frame is not the "real" father of its environment. This type of node must be marked so that it is saved from the "cut" predicate, which eliminates all the backtrack points up to and including its "real" parent. The execution continues until the creation of node N4 which is non-deterministic and we cannot apply TRO.

In general, the situation in which TRO applies is not necessarily recursive. More appropriately, it should be called "last call optimisation" (LCO). In applying LCO there is a possibility that the old and new environments are of different sizes, so it is difficult to swap the two as a complete segment of the runtime stack between the environments would have to be relocated. In WUP, the restriction is such that the old and new environments must be of the same size in order to apply LCO.

Note that LCO can be applied even before the call successfully returns; if the optimised call ultimately fails, it will not affect the state of computation since it was deterministic and it is the last call in its parent clause; if its parent clause is deterministic, then the call to its parent clause fails; otherwise, its parent clause's stack node is popped and rebuilt for the next alternative(s). In the case of DCO, we have to wait until the call exit. So, LCO has a bigger benefit for systems with limited amount of memory.

4.4. An Improved Version of the ABC Algorithm

With the introduction of all the necessary basic concepts, we now turn our discussion back to our ABC algorithm which incorporates all the above-mentioned optimisations. In WUP, this algorithm is implemented as the built-in "prove" predicate. The state of execution is characterised by a set of machine registers:

Cur-frame	a pointer to the control stack frame for the current node (the current node is the head of the list of all the parents on a particular branch of the proof tree).
Cur-env	a pointer to the environment of the current node.
Cur-call	a pure code pointer to the current call that is selected for execution.
Cur-proc	a pure code pointer to the procedure containing clauses that possibly-match the current call.
Cur-module	a pointer to the module to which the current call belongs.
Cur-back	a stack frame pointer to the most recent backtrack point (head of the list of all backtrack points).

In figure 4.7, we present the complete ABC algorithm in an Ada-like language. All necessary support routines are briefly described in Appendix D.

4.5. Unification Algorithm

The unification algorithm used in WUP is no different from previous Waterloo Unix Prolog implementations. It does not include the "occurs check". So, for unifying infinite structures, our algorithm loops until available storage is exhausted, and then the execution will be aborted.

Since lists and functors are implemented as different data structures in WUP, we have more cases for testing whether two terms are unifiable. For efficiency, our unification routine is table-driven (see figure 4.8). Based on this table, the unification routine can now be condensed into fewer and more

```
module ABC is
   StackRange = 1..MAX_STACK_SIZE ;
   RT_top, Copy_top, Trail_top : StackRange ;
   Cur_frame, Cur_back : FramePtr ;
   Cur_env, Init_env : EnvPtr ;
   Cur_proc : Clause ;
   Cur_call : GoalCall ;
   Cur_cg : Procedure ;
   Cur_module : ModulePtr ;

procedure select () returns ( Boolean ) ; external ;
procedure son () returns ( Boolean ) ; external ;

procedure solve ( M: ModulePtr ; Q: Clause ) returns ( Boolean ) ;
begin
   ( Cur_frame, Init_env) := push-stack-node( M, Q, DET) ;
   ( FATHER[ Cur_frame ], CALL[ Cur_frame ]) := ( NIL, NIL ) ;
   ( Cur_proc, Cur_call, Cur_env, Cur_module, Cur_back ) :=
      ( Q, NIL, Init_env, M, NIL ) ;

A:  if select() then
      ( Cur_cg, Cur_module ) := clause-generator( Cur_module, Cur_call ) ;
   else
      return ( true ) ;
   endif ;
B:  if son() then
      goto A ;
   endif ;
C:  if Cur_back = NIL then
      return ( false ) ;
   else
      undo-binding( RESET[ Cur_back ] ) ;
      pop-copy-stack( COPY[ Cur_back ] ) ;
      pop-stack-node( Cur_back ) ;
      ( Cur_frame, Cur_call, Cur_cg, Cur_module ) :=
         ( FATHER[ Cur_back ], CALL[ Cur_back ], CL-GEN[ Cur_back ],
           MODULE[ Cur_back ] ) ;
      Cur_env := env-ptr-of( Cur_frame ) ;
      Cur_back := BACK[ Cur_back ] ;
      goto B ;
   endif ;
end solve ;
endmodule.
```

Figure 4.7a The ABC Algorithm (solve)

```
procedure select () returns ( Boolean ) ;
begin
   % forward selection step
   Cur_call := first-call-of( Cur_proc ) ;
   while Cur_call = NIL loop
      % popping deterministic node
      if Cur_frame > Cur_back then
         pop-stack-node( Cur_frame ) ;
      endif ;
      % backward selection step
      Cur_call := next-call-of( CALL[ Cur_frame ] ) ;
      Cur_frame := FATHER[ Cur_frame ] ;
      % reach the root of the proof tree
      if Cur_frame = NIL then
         return ( false ) ;
      endif ;
   endloop ;
   Cur_env  := env-ptr-of( Cur_frame ) ;
   Cur_module := MODULE[ Cur_frame ] ;
   return ( true ) ;
end select ;
```

Figure 4.7b The ABC Algorithm (select)

manageable cases.

4.6. Structure-copying Algorithm

As described previously in figure 4.5, a copy of a structured object containing variable(s) is made on the copy stack whenever it unifies with a free variable. There are cases where the new copy also contains free variable(s). The question now is how to maintain the equivalence between a variable with an instance in the copy stack and another instance in the runtime stack (environment). As long as the two instances are linked by some pointer, the interpretation will be correct. But, as mentioned before, in order to apply the storage optimisation techniques, we cannot leave dangling pointers in the

```
procedure son () returns ( Boolean ) ;
   tframe : FramePtr ;              tenv : EnvPtr ;
   tproc : Clause ;                 TRO : Boolean ;
   copy : StackRange := Copy_stack_top ;
   reset : StackRange := Trail_stack_top ;
begin
   while Cur_cg <> NIL loop
      ( tproc, Cur_cg ) := next-clause( Cur_cg ) ;
      if Cur_cg = NIL then
         set-binding-limit( Cur_back, COPY[ Cur_back ] ) ;
         ( tframe, tenv ) := push-stack-node( Cur_module, tproc, DET ) ;
      else
         set-binding-limit( tframe, copy ) ;
         ( tframe, tenv ) := push-stack-node( Cur_module, tproc, NDET ) ;
      endif ;
      TRO := next-call-of( Cur_call ) = NIL and is-det-node( tframe ) and
           env-size-of( Cur_env ) = env-size-of( tenv ) and
           Cur_module = MODULE[ Cur_frame ] ;
      if TRO then
         swap-environment( Cur_env, tenv ) ;
         ( Cur_env, tenv ) := ( tenv, Cur_env) ;
      endif ;
      if unify( Cur_call, Cur_env, head-predicate-of(tproc), tenv ) then
         if not is-det-node( tframe ) then
            ( BACK[ tframe ], COPY[ tframe ], CL-GEN[ tframe ],
              RESET[ tframe ] ) := ( Cur_back, copy, Cur_cg, reset ) ;
         endif ;
         if TRO then
            pop-stack-node( tframe ) ; set-not-real-father( Cur_frame ) ;
         else
            ( MODULE[ tframe ], CALL[ tframe ], FATHER[ tframe ] ) :=
               Cur_module, Cur_call, Cur_frame ) ;
            Cur_frame := tframe ;
         endif ;
         ( Cur_proc, Cur_env ) := ( tproc, tenv ) ;
         return ( true ) ;
      else
         undo-binding( reset ) ; pop-stack-node( tframe ) ; pop-copy-stack( copy ) ;
      endif ;
   endloop ;
   return ( false ) ;
end son ;
```

Figure 4.7c The ABC Algorithm (son)

CALL / HEAD	Free Var.	Void Var.	Int. Const	Atom Const	Char Const	End List	Const List	Var. List	Const Func	Var Func
Free Variable	FV	S	AH	AH	AH	AH	AH	CH	AH	CH
Void Variable	S	S	S	S	S	S	S	S	S	S
Integer Constant	AC	S	SC	F	F	F	F	F	F	F
Atom Constant	AC	S	F	SC	F	F	F	F	F	F
Character Constant	AC	S	F	F	SC	F	F	F	F	F
End List Constant	AC	S	F	F	F	SC	F	F	F	F
Constant List	AC	S	F	F	F	F	UL	UL	F	F
Variable List	CC	S	F	F	F	F	UL	UL	F	F
Constant Functor	AC	S	F	F.	F	F	F	F	UF	UF
Variable Functor	CC	S	F	F	F	F	F	F	UF	UF

F	- always Fail	UL	- Unify Lists
S	- always Succeed	UF	- Unify Functors
FV	- Free Variables	CH	- Copy to Head
AH	- Assign to Head	CC	- Copy to Call
AC	- Assign to Call	SC	- Simple Comparison

Figure 4.8 Unification Table

environments. One general assumption is that all pointers within the environments are always oriented downward, from top to bottom of the runtime stack. To guarentee correctness of those optimisations, there is no doubt that we should orient the pointers from the environments to the copy stack. For uniformity, we would also orient the pointers downward whenever two free

```
procedure unify ( C: Functor; CE: EnvPtr; H: Functor; HE: EnvPtr ) returns ( Boolean ) ;
begin
   C := dereference( C, CE ) ;
   H := dereference( H, HE ) ;
   if H = C then return ( true ) ; endif ;
   case Unify_table[ TAG[ H ], TAG[ C ] ] is
      when F => return ( false ) ;
      when S => return ( true ) ;
      when FV =>
         if H > C then
            ( TAG[ H ], VAL[ H ] ) := ( REF, C ) ; record-binding( H ) ;
         else
            ( TAG[ C ], VAL[ C ] ) := ( REF, H ) ; record-binding( C ) ;
         endif ;
      when AH =>
         H := C ; record-binding( H ) ;
      when AC =>
         C := H ; record-binding( C ) ;
      when UL =>
         if not unify( LIST_HEAD[ C ], CE, LIST_HEAD[ H ], HE ) or not
            unify( LIST_TAIL[ C ], CE, LIST_TAIL[ H ], HE ) then
            return ( false ) ;
         endif ;
      when UF =>
         if FUNC_NAME[ C ] = FUNC_NAME[ H ] and then
            ARITY[ C ] = ARITY[ H ] then
            for i in ARITY[ C ] loop
               if not unify( argument-of( i, C ), CE, argument-of( i, H ), HE ) then
                  return ( false ) ;
            endloop ;
         else
            return ( false ) ;
         endif ;
      when CH =>
         new-copy-to( C, CE, H ) ;
      when CC =>
         new-copy-to( H, HE, C ) ;
      when SC =>
         return ( VAL[ H ] = VAL[ C ] ) ;
   endcase ;
   return ( true ) ;
end unify ;
```

Figure 4.9 Unification Algorithm

variables in the copy stack are unified. Our assumption is that the pointers for variable bindings are always oriented from top to bottom in the runtime or copy stack and from runtime to copy stack. Hence, the danger of dangling pointers after optimisation is avoided.

```
procedure new-copy-to ( From: Term; E: EnvPtr; To: Term ) ;
begin
   From := deference( From, E ) ;
   % don't copy existing term on the copy stack
   if in-copy-stack( From ) then
      To := From ; return ;
   elsif From = To then return ; endif ;
   case TAG[ From ] is
      when Free Variable =>
         if From > To then
            ( TAG[ From ], VAL[ From ] ) := ( REF, To ) ;
            record-binding( From ) ;
         else
            ( TAG[ To ], VAL[ To ] ) := ( REF, From ) ;
            record-binding( To ) ;
         endif ;
      when Constants or End List or Constant Functor or Constant List =>
         To := From ;
         record-binding( To ) ;
      when Variable List =>
         new-copy-to( LIST_HEAD[ From ], E, LIST_HEAD[ To ] ) ;
         new-copy-to( LIST_TAIL[ From ], E, LIST_TAIL[ To ] ) ;
      when Variable Functor =>
         ( FUNC_NAME[ To ], ARITY[ To ] ) :=
               ( FUNC_NAME[ Frome ], ARITY[ From ] ) ;
         for i in ARITY[ From ] loop
            new-copy-to( argument-of( i,From ), E, argument-of( i,To ) ) ;
         endloop ;
    endcase ;
end new-copy-to ;
```

Figure 4.10 Copying Algorithm

The copying algorithm is a simple recursive routine that uses the tag field of the copied object to determine whether a new copy should be made, or just a simple assignment should be done (see figure 4.10). The algorithm does not perform full copying, i.e. the constant terms (scalar or compound) are shared. One observation about copying is that we are only concerned with the pure-code terms, i.e. the static read-only code generated at compile-time. Once a copy of a pure-code term is made, this new copy will be shared. It is crucial to identify a term which is already a copy of some pure-code term because without such a distinction a program linear in space can be turned into one that is quadratic.

One final remark about this implementation of the structure-copying algorithm: when dealing with infinite terms, as in the unification algorithm, this algorithm can go into an infinite loop until the copy stack overflows.

4.7. Built-in Predicates

In implementing a Prolog system, the majority of the effort is on the "cosmetic" features -- the built-in primitives necessary for making Prolog a usable programming environment. Some are "extralogical" in order to make Prolog more efficient (e.g. cut). These primitives can be classified into the following main categories: arithmetic, control, i/o, meta-logical, database and relational. We give a few example predicates for each of these categories:

arithmetic	add, mul, mod
control	cut, or, prove
i/o	read, write, get, put
meta-logical	is_var, is_functor, is_atom
database	assert, retract, clause

relational lt, gt, eq

The above lists only a very small number of built-in primitives supported in WUP. They will not be described here in full detail, however, we will discuss the implementation of some of the more interesting ones: cut, prove and clause.

cut or '!'

This predicate has a side-effect on the selection of execution alternatives. It will remove all the alternatives up to and including the clause to which it belongs. For the implementation, it makes deterministic all the nodes between the current node (the call to "cut") and its parent node. Because of TRO, the parent node may not be the "real" father, and should be left unchanged. The implementation of "cut" in WUP is given below (note that the variables Cur-back and Cur-frame are those declared in the ABC algorithm, see figure 4.7a):

```
% remove all choice points between the current node
% and its parent node
%
while ( Cur-back > Cur-frame ) loop
   set-det-node( Cur-back ) ;
   Cur-back := BACK[ Cur-back ] ;
endloop ;

if is-real-father( Cur-frame ) and  Cur-back = Cur-frame then
   set-det-node( Cur-frame ) ;
   Cur-back := BACK[ Cur-back ] ;
endif ;
```

After the application of a "cut", any variable bindings that were created between the current node and its parent node are now frozen, i.e. they will not be reset upon backtracking. Hence, their binding records on the trail stack can be removed. This could provide a marginal saving in storage for a very deep recursive call where "cuts" are used to remove alternatives.

prove(Goal)

In WUP, the internal representation of a goal call and a normal Prolog term is slightly different; a goal call has some extra information about the matched procedure and the next call in the same clause. The structure representing the goal call "prove(Goal)" cannot be modified, we must find some way to construct another structure representing the goal call "Goal" and to insert it between the the goal call "prove" and the next goal call, so that the new "Goal" call will be executed next after the successful return of the predicate "prove". One solution is to make a copy of the goal call "prove(Goal)", modify the new copy's next call pointer to the new goal call for "Goal" and the new goal call's next call pointer to the next goal call of the original "prove(Goal)". Figure 4.11 shows an example of how the data structure is adjusted at runtime.

clause(Module, Head, Body)

Given a term "Head", "clause" will try to retrieve from the database of the module "Module" a clause whose head is unifiable with "Head" and its body is then unified with "Body". Later upon backtracking, this predicate will be retried on the next clause which has the same properties. This predicate fails if nothing in the database is unifiable with "Head".

One immediate problem is how to index through all the database clauses without explicit references. One solution is to allocate a new temporary variable on top of the current environment, and use it as the "reference" to the clause found. This variable should never be reset upon backtracking, so it not recorded on the trail stack. A new problem now is how to distinguish the first, second or

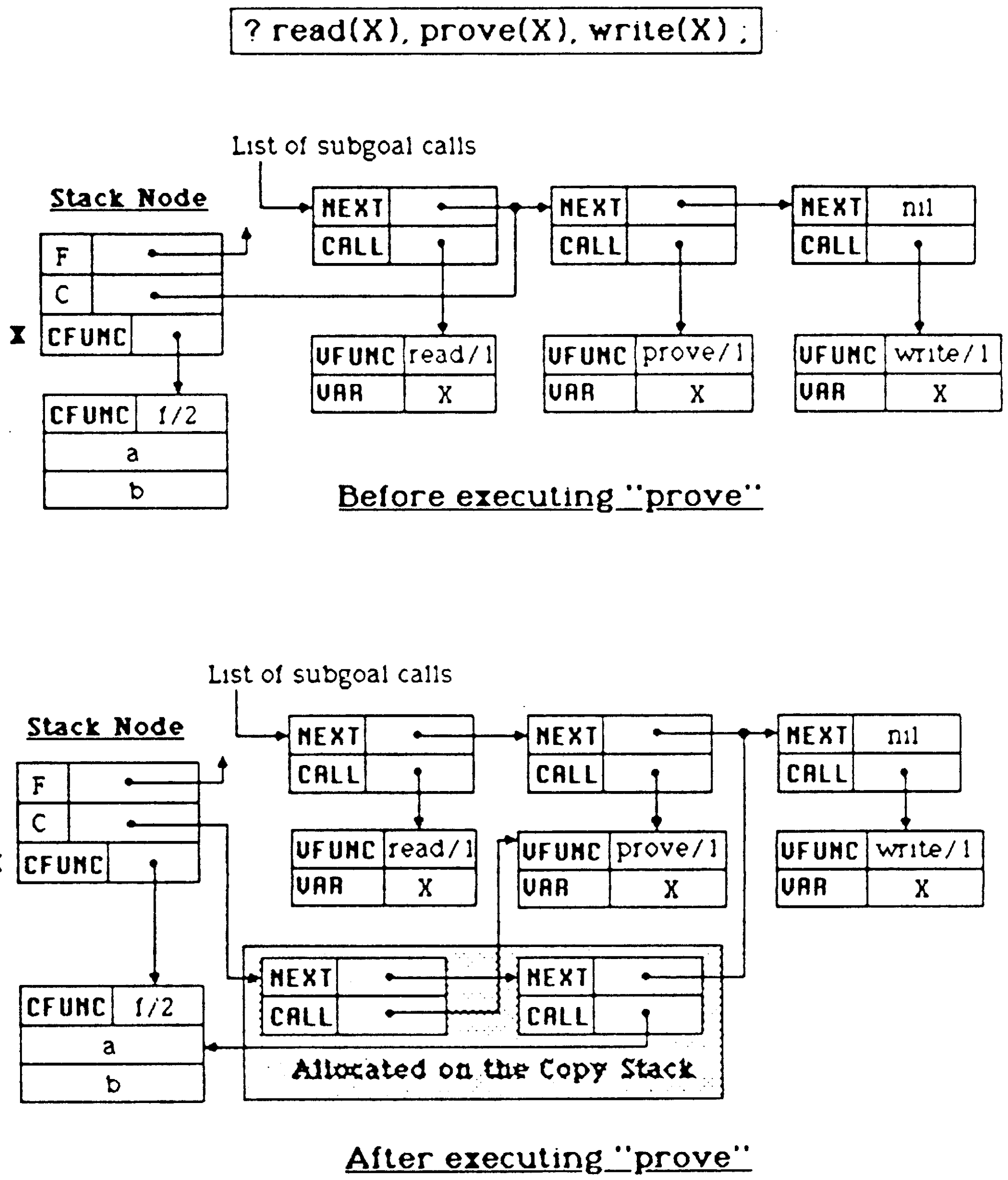


Figure 4.11 Execution of Prove Predicate

subsequent invocations to "clause/3". When it is first called, the temporary variable (or reference) does not exist yet. In WUP, a state variable is declared in the ABC algorithm and is used to indicate the state of computation. If a call to 'clause/3" is made by backtracking, the state variable is set and we know this call cannot be the first invocation.

4.8. Trace Facility

The debugging facility in WUP is based on the "box" model [Byr80] which describes the execution of a Prolog clause in terms of four different states: CALL, EXIT, REDO and FAIL. In WUP, we add an extra message "MORE" for indicating that the current call succeeds, and has alternatives (i.e. it is choice point). These states can be easily incorporated into the ABC algorithm. Their relationship with the ABC algorithm are given below:

CALL	after every successful selection step (forward or backward).
EXIT	just before every backward selection step, i.e. the current call is the last call in the current node.
REDO	at every backtracking step.
FAIL	after every unsuccessful call to "son()" generation step.
MORE	at every successful return from "son()" when the "clause-generator" is not empty.

Apart from giving the state of execution, the trace information also indicates the location in the proof tree of every state by a unique tuple, (tree level, node number). The implicit query "?" is at (0,0). For every "CALL", the node number is increased by 1. The tree level of any node is the tree level of its immediate parent plus 1. (Note: During debugging, no optimisation is performed.) Both the tree level and node number are stored in the stack frame; thus, it is straight-forward to determine the location of each state.

Chapter 5
Conclusions

WUP has been fully operational since January 1984. It has been used as a teaching tool for the course CS486 (Introduction to Artificial Intelligence) at University of Waterloo. During this period, many of the more serious system bugs have been fixed. The concepts of "Modules" and "Programming Environment" in Prolog have been studied. As a first attempt in this direction, the results to-date have been satisfactory. We have now learned more about the feasibility of building a Prolog environment in Prolog. Our next step is to evaluate WUP's user interface and its usefulness for real programming tasks. In the following sections, we show one interesting test program for our unification algorithm, provide the results of comparisons with CProlog [Per83], summarise the central ideas in this implementation (and the things that we would do differently next time) and finally state a few possible short-term and long-term extensions for WUP and Prolog in general.

5.1. Unification Test

Figure 5.1 shows a list construction and traversal program in WUP. The program is written in Lisp-style of "car-and-cdr" operations. The result of this program demonstrates the ease of constructing two-way cyclic lists in Prolog. Its main idea is to illustrate the power of "unification" in Prolog. (Note that the unification step is represented by the predicate "eq".)

At the end of execution of this program, the variable X contains the term "[[a21 | [a22 | b22]] | [[a22 | b22] | b12]]" with only one copy of the term "[a22 | b22]", which implies that there is a cycle in X. Another view of looking at this program is as a set of recursive equations with "eq" being the "=" operator. Then, the term that X represents is equivalent to the set of equations { X = [Z|Y], Z = [a21|W], Y = [W|B21], W = [a22|b22] }. To construct such a structure in Lisp is impossible without destructive assignment [Mor80]. Most Lisp implementations provide the list-structure-altering primitives for modifying the "car" and "cdr" fields -- "rplaca" and "rplacd". However this destroys the mathematical elegance of Lisp's lambda calulus foundation.

```
eq( X, X ) ;
car( [X|_], X ) ;
cdr( [_|Y], Y ) ;

test <-
   eq( X, [Z|Y] ),
   eq( Y, [W|b12] ),
   eq( Z, [a21|W] ),
   eq( W, [U|V] ),
   car( X, Z1 ), cdr( Z1, Z2 ), eq( Z2, [a22|b22] ),
   cdr( X, Y1 ), car( Y1, Y2 ), write( Y2 ), nl
   ;

? test ;
[a22|b22]
yes
```

Figure 5.1 A List Construction and Traversal Program

5.2. Comparative Studies

At the time of writing, CProlog is the only other Prolog system available at Waterloo on Unix. Initially, we intended to include comparisons of both execution time and storage usage on a few standard test programs, but there is no obvious way to determine the storage usage (in particular of the main stacks) in CProlog. Thus, we can provide only the statistics on execution time for both systems. One thing to keep in mind is that these two systems are very different. CProlog is an interpreter designed and implemented for pure speed and practical purposes; WUP is an experimental system for testing new concepts. Although both systems are written in C on Unix, WUP is designed for its modularity and extendability. It is not surprising that CProlog is about twice as fast as WUP.

(Time in seconds)			
Programs	Input size	CProlog	WUP
(1) fibonacci	18	42.583	57.850
(2) naive reverse	30	0.283	0.617
(3) quicksort	20	0.133	0.267
(4) hanoi	10	2.099	2.833
(5) hanoi	15	stack overflows	96.003

Figure 5.2 CProlog versus WUP

In figure 5.2, we list the results of comparisons on four test programs using the "cputime" predicate on CProlog and "time" predicate on WUP (see Appendix E for details on the test programs). For the programs (2) and (3), CProlog is about two times faster than WUP because it uses structure-sharing (SS), while WUP uses structure-copying (SC). It is known that the SS approach performs better than the SC approach in constructing new terms at execution time. The programs (2) and (3) involve many list construction operations, while (4) requires none. As shown in program (4), CProlog is marginally faster than WUP. However, it runs out of available stack space for the same program with bigger input size. WUP still performs reasonably well with TRO.

Our conclusions about these comparisons are: (1) the execution efficiency of any Prolog systems depend mainly on the types of input programs and their chosen schemes for constructing new terms, assuming that they are all implemented in the same language on the same machine; (2) the main issue on Prolog implementation lies with storage optimisation; no matter how fast a system can run, it cannot continue its execution if the available storage is exhausted.

5.3. Summary

Many of the design decisions are influenced by experience with previous implementations and comments from our users. The implementation and optimisation techniques used in WUP are not drastically different from other Prolog implementations except at the lowest level of details. The organisation and design of WUP has been much directed towards our original aims: portability, modularity and extendability. The following is a summary of some interesting features in WUP:

Compiling

An interpreter provides good debugging facilities for program development; a compiler produces fast executable code for production. Our approach is a compromise between pure interpretation and pure compilation. Compiling Prolog into virtual machine code has also been discussed recently in [Bow83,Bal83] and their objectives are portability and efficiency.

Database

The use of the existing file system (Unix) to organise the program database simplifies maintainance of very large programs. Integrating Prolog with a relational database machine has also been suggested by [Bal83] for efficiently storing and retrieving large amounts of facts. All these research directions will in some way affect future extensions of WUP. Our current design is an attempt to make Prolog more usable for large programming projects.

Debugging

The "box" model [Byr80] provides a simple but adequate view of watching Prolog at work. With good debugging and separate compilation facilities, large Prolog program can be tested and assembled. The design of WUP's debugging facility comes mostly from discussions with students that have used other Prolog systems.

Modules

The new notion of modules in Prolog is the most attractive part, and that has also generated further research to investigate its semantics and practicality in logic programming. This notion, we believe, will become very important in the area of practical logic programming. Similiar approaches have also been proposed recently in [Bow83,Ben80,Bal83,Chi83,Fur83]. The module defined in MProlog [Ben80] and Micro-Prolog [Cla81] are basically for the purpose of reusable software and separate compilation.

ABC algorithm

The ABC algorithm has been used in the previous two implementations of Waterloo Unix Prolog. Its succinct description of Prolog's underlying execution mechanism is one of the major contributions to the continuing effort of improvements. We hope our introduction of optimisations will not destroy the clarity of the original.

Unification

The increase in the number of basic types (e.g. integer, string, etc.) in Prolog can complicate the unification algorithm. A table-driven unification routine can reduce the total number of cases into a few manageable ones. This is particularly suitable for hardware implementation. A similar approach has also been proposed in [Bal83].

Structure-copying

As an alternative to the structure-sharing approach, the structure-copying approach has other advantages. As noted by Bowen [Bow83], structure-copying may give a better locality of references for a virtual memory based Prolog implementation. Structure-copying is also easier to be implemented on machines that cannot store two addresses in a single machine word.

Optimisations

Tail recursion (TRO) and deterministic call (DCO) optimisations have become almost absolutely necessary in any future Prolog implementations. In practice, if we can implement TRO efficiently, we can promote recursive rather than iterative style of programming (Note that the latter requires "destructive" assignment which cannot be done in Prolog in general). As database machines become available, the technique of clause indexing becomes less important. For a large database application, software indexing becomes impractical.

Memory Layout

The major assumption about pointer orientation in the copy and runtime stack has somewhat affected the memory layout of our actual implementation. One simplication that can be easily achieved is to allocate the two stacks, copy and runtime, in the same storage area with the runtime stack on top of the copy stack; under this condition, the pointers are always oriented downward, and we can guarentee our original assumption.

What to do differently next time

The unification and structure-copying algorithms employed in this implementation have one drawback in that they are both recursive; in Unix this has the problem of causing a "memory fault" when operating on infinite terms. An alternative approach is to use an explicit "auxiliary" stack for maintaining nested or partially finished structure terms. In that case, we can replace recursion by iteration and raise an exception whenever the "auxiliary" stack overflows. This could have a slight improvement on the speed of execution of these two algorithms because it involves fewer procedure calls.

Another thing that to do differently next time is the selection of built-in primitives. We have no known guiding principle prior to our implementation on what the minimal but usable set of primitives is required in order to support a programming environment in Prolog. Our approach is ad hoc. Until there is a better understanding of what Prolog modules are and how they are used, we cannot pinpoint the real problem in such an implementation. In the next version of WUP, we can provide some insight which will help contribute to the development of a complete Prolog programming environment.

5.4. Future Works and Extensions

Prolog is still an evolving programming language. Its future success relies on its suitability for large programming projects. In this document, we gave an account of the design and implementation issues of a programming environment for Prolog, which is a step forward in the direction of making Prolog more usable and practical. Throughout our discussions, we have emphasised the importance of the concept, "Modules in Prolog". To fully appreciate the usefulness of this concept, we propose in the following, a list of future plausible short-term and long-term extensions in WUP or Prolog.

Short-term Extensions

I) Modules in Prolog

a) interface to non-Prolog code -- by defining the protocols of communication among modules, a Prolog module can communicate with non-Prolog modules.

b) generalise the search for a proof -- the search path can be organised into a tree hierarachical structure, e.g. like Smalltalk, and its configuration should be user-definable.

c) query-the-user -- the user can be viewed as another module which observes certain properties; hence, a non-provable fact can be reflected to this module for possible solutions.

II) Environment Supports

a) more extensive editing facilities -- a syntax-directed editor with an integrated debugger is a powerful tool for watching Prolog at work while developing programs.

b) a more elaborate file system specially tailored for Prolog sources, code and databases. Avoiding the distinction of these objects, a user operates directly only on modules or predicates.

c) an automatic garbage-collection facility -- this is the ultimate solution to the problem of stacks overflow.

Long-term Extensions

I) Parallelism -- by defining the communication and synchronisation primitives for modules, we can view modules as independent execution units and run them asynchronously.

II) Heuristic Control -- without modifying the Horn clause semantics of Prolog, we provide a way of including a set of Horn clauses for describing the execution order of other sets of clauses in order to speed up the search of proofs.

III) Relational Database Machine Interface -- again, based on the concept of module, we define the lazy and eager evaluation primitives for accessing a large database which is encapsulated as a module.

References

[Aho77] Aho, A.V. and Ullman, J.D. (1977), *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts.

[Bal83] Ballieu, G. (1983), A Virtual Machine to Implement Prolog, *Proceedings of Logic Programming Workshop*, Algarve, Portugal, pp. 40-52.

[Bat73] Battani, G. and Meloni, H. (1973), Interpreteur du language de programmation Prolog, Groupe d'Intelligence Artificielle, University d'Aix Marseille, Luminy, France.

[Ben80] Bendl, J., Koves, P., and Szeredi, P. (1980), The MPROLOG System, *Proceedings of the Logic Programming Workshop*, July, Debrecen, Hungary, S.A. Tarnlund (ed.), pp. 201-209.

[Ber73] Bergman, M. and Kanoui, H. (1973), Application of mechanical theorem proving to symbolic calculus, *Third International Symposium on Advanced Computing Methods in Theoretical Physics, C.N.R.S.*, Marseille, France.

[Bow83] Bowen, D.L., Byrd, L.M., and Clocksin, W.F. (1983), A Portable Prolog Compiler, *Proceedings of the Logic Programming Workshop*, Algarve, Portugal.

[Bow82] Bowen, K.A. and Kowalski, R.A. (1982), Amalgamating Language and Metalanguage in Logic Programming, *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, pp. 153-172.

[Bru76] Bruynooghe, M. (1976), An interpreter for predicate logic programs: part 1, Report CW10, Applied Math and Programming Division, Katholieke University,, Leuven, Belgium..

[Bru82] Bruynooghe, M. (1982), The Memory Management of Prolog Implementation, *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, pp. 83-98.

[Bru82a] Bruynooghe, M. (1982), A Note on Garbage Collection in Prolog Interpreters, *Proceedings of the First International Logic Programming Conference*, September, Marseille, France, pp. 52-55.

[Bru83] Bruynooghe, M. (1983), Some Reflexions on Implementation Issues of Prolog, *Proceedings of the Logic Programming Workshop*, Algarve, Portugal, pp. 1-6.

[Byr80] Byrd, L. (1980), Understanding the Control Flow of PROLOG Programs, *Proceedings of the Logic Programming Workshop*, July, Debrecen, Hungary, S.A. Tarnlund (ed.), pp. 127-138.

[Che84] Cheng, M.H.M. and Goebel, R. (1984), Waterloo Unix Prolog Implementation Manual, Logic Programming Group, University of Waterloo, Waterloo, Ontario [in preparation].

[Chi83] Chikayama, T. (1983), ESP - Extended Self-contained Prolog - as a Preliminary Kernel Language of Fifth Generation Computers, *New Generation Computing* **1**(1), Institute for New Generation Computer Technology, pp. 11-24.

[Chi84] Chikayama, T. (1984), ESP Reference Manual, *ICOT Technical Report TR-044*, Institute for New Generation Computer Technology, Tokyo, Japan, Feburary.

[Cla81] Clark, K.L. and McCabe, F.G. (1981), Micro-Prolog Reference Manual, Logic Programming Associates, London, England.

[Cla82] Clark, K.L., McCabe, F.G., and Gregory, S. (1982), IC-Prolog Language Features, *Logic Programming*, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, pp. 253-266.

[Col73] Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. (1973), Un System de Comunication Homme-machine en Francais Rapport, Groupe d'Intelligence Artificielle, Universite d'Aix Marseille, Luminy, France.

[Con83] Conery, J.S. (1983), The AND/OR Process Model for Parallel Interpretation of Logic Programs, Dept. of Information and Computer Science, University of California, Irvine, California, U.S., June.

[DOD80] DOD, (February, 1980), Stoneman: Requirements for Ada Programming Support Environment.

[Dar78] Darvas, F., Futo, I., Szeredi, J., Bendl, J., and Koves, P. (1978), A PROLOG-based drug design system (Hungarian), *Proc. of Conf. Programming Systems'78*, Szeged, Hungary, pp. 119-126.

[Egg82] Eggert, P.R. and Schorre, D.V. (1982), Logic Enhancement: A Method for Extending Logic Programming Languages, *ACM Symp. on LISP and Functional Programming*, August, Pittsburgh, Pennsylvania, pp. 74-80.

[Fel79] Feldman, S.I. (1979), Make - A program for Maintaining Computer Programs, *UNIX Programmer's Manual, seventh edition* **2A**, Bell Laboratories, Murray, New Jersey.

[Fer81] Ferguson, R.J. (1981), A Prolog Interpreter for the Unix Operating System, M.Math. dissertation, Department of Computer Science, University of Waterloo.

[Feu83] Feuer, A. (1983), Building Libraries in Prolog, *IJCAI'83* **1**, Karlsruhe, West Germany, pp. 547-552.

[Fur83] Furukawa, K., Nakajima, R., and Yonezawa, A. (1983), Modularization and Abstraction in Logic Programming, *New Generation Computing* **1**(2), pp. 169-178.

[Gal78] Gallaire, H. and Minker, J. (1978), *Logic and Data Bases*, Plenum Press, New York.

[Gut77] Guttag, J.V. (1977), Abstract Data Types and the Development of Data Structures, *ACM Communications* **20**(6), pp. 396-404.

[Hoa78] Hoare, C.A.R. (1978), Communicating Sequential Processes, *ACM Communications* **21**(8), pp. 666-677.

[Hog82] Hogger, C.J. (1982), Concurrent Logic Programming, *Logic Programming*, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, pp. 253-266.

[ICO82] ICOT, (1982), Outline of Research and Development Plans for FGCS, ICOT Research Center, Minato-ku, Tokyo, May.

[Ing78] Ingalls, D. (1978), The Smalltalk-76 Programming System: Design and Implementation, *5th Annual Symposium on Principles of Programming Languages*, January, pp. 9-16.

[Joh75] Johnson, S.C. (1975), YACC: Yet Another Compiler Compiler, Technical Report No.32, Computing Science, Bell Laboratories , Murray Hill, N.J..

[Jon80] Jones, S. (1980), Structured Programming Techniques in PROLOG, *Proceedings of the Logic Programming Workshop*, July, Debrecen, Hungary, S.A. Tarnlund (ed.), pp. 322-333.

[Kah82] Kahn, K.M. (1982), Intermission - Actors in PROLOG, *Logic Programming*, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, pp. 213-228.

[Ker78] Kernighan, B.W. and Ritchie, D.M. (1978), *The C programming language*, Prentice-Hall, Englewood Cliffs, New Jersey.

[Lis74] Liskov, B.H. and Zilles, S.N. (1974), Programming with Abstract Data Types, *ACM SIGPLAN Notices* **9**(4), April, pp. 50-59 [SIGPLAN Symposium on Very High Level Languages].

[Mal83] Malcolm, M. and Didur, P. (1983), Waterloo Port User Guide, Software Portability Group, University of Waterloo, Waterloo, Ontario.

[Mar77] Markusz, Z. (1977), How to design variants of flats using programming language PROLOG, based on mathematical logic, *Proc. of IFIP 77*, Toronto, pp. 885-890.

[Mel82] Mellish, C.S. (1982), An Alternative to Structure-Sharing in the Implementation of a Prolog Interpreter, *Logic Programming*, A.P.I.C Studies in Data Processing, K.L. Clark and S.A. Tarnlund (eds.), Academic Press, New York, pp. 99-106.

[Miz83] Mizoguchi, F. (1983), PROLOG Based Expert System, *New Generation Computing* **1**(1), Institute for New Generation Computer Technology, pp. 99-104.

[Mor80] Morris, F.L. and Schwarz, J.S. (1980), Computing Cyclic List Structure, *Proceedings of the 1980 Lisp Conference*, Stanford, USA., pp. 144-153.

[Mos80] Moss, C.D.S. (1980), The comparison of several PROLOG systems, *Proceedings of the Logic Programming Workshop*, July, Debrecen, Hungary, S.A. Tarnlund (ed.), pp. 198-200.

[Ng82] Ng, T.Y. (1982), Prolog Implementation, M.Math. dissertation, Department of Computer Science, University of Waterloo, Waterloo, Ontario.

[Ona82] Onai, R., Shimizu, H., Ito, N., and Masuda, K. (1982), The Proposal of Prolog Machine based on Reduction Mechanism, First Research Laboratory, Research Center, ICOT.

[Per83] Pereira, F., Warren, D.H.D., Byrd, L., and Pereira, L.M. (1983), CProlog User's Manual Version 1.2, SRI International, Menlo Park, California.

[Poo84] Poole, D.L. (1984), The Theory of Modules as Theories (or generic types in PROLOG), Logic Programming and Artificial Intelligence Group, University of Waterloo, Waterloo, Ontario [in preparation].

[Rob77] Roberts, G.M. (1977), An implementation of PROLOG, M.Math. dissertation, Computer Science Department, University of Waterloo, April.

[Rob65] Robinson, J.A. (1965), A Machine Oriented Logic Based on the Resolution Principle., *ACM Journal* **12**(1), pp. 23-41.

[Rou72] Roussel, P. (1972), Definition et traitement de l'egalite formelle en demonstration automatique, These 3me. cycle, UER de Luminy, Marseille, France.

[San78] Sandewall, E. (1978), Programming in the Interactive Environment: The LISP Experience, *ACM Computing Surveys* **10**(1), March, pp. 35-71.

[Sha83] Shapiro, E. and Takeuchi, A. (1983), Object Oriented Programming in Concurrent Prolog, *New Generation Computing* **1**(1), pp. 25-48.

[Sha83a] Shapiro, E.Y. (1983), *Algorithmic Program Debugging*, MIT Press [Ph.D. thesis, Yale University, May 1982].

[Tre82] Treleaven, P.C., Brownbridge, D.R., and Hopkins, R.C. (1982), Data-Driven and Demand-Driven Computer Architecture, *ACM Computing Surveys* **14**(1), March, pp. 93-143.

[War76] Warren, D.H.D. (1976), Generating conditional plans and programs, *Proc. AISB Summer Conference, 1976*, Edinburgh, pp. 344-354.

[War77] Warren, D.H.D. (1977), Implementing Prolog - Compiling Predicate Logic Programs, Research Report 39/40, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland.

[War79] Warren, D.H.D., Pereira, L.M., and Pereira, F. (1979), User's Guide to DECsystem-10 Prolog, Occasional Paper 15, Department of Artifical Intelligence, University of Edinburgh.

[War80] Warren, D.H.D. (1980), An Improved Prolog Implementation which Optimises Tail Recursion, *Proceedings of the Logic Programming Workshop*, July, 1980, Debrecen, Hungary, S.A. Tarnlund (ed.).

[Wis82] Wise, M.J. (1982), A PARALLEL PROLOG: the construction of a data driven model, *ACM Symp. on LISP and Functional Programming*, August, 1982, Pittsburgh, Pennsylvania, pp. 56-66.

[van81] vanEmden, M.H. (1981), An Algorithm for Interpreting Prolog Programs, Research Report CS-81-28, Department of Computer Science, University of Waterloo.

[van82] vanEmden, M.H. and Filho, G.J. de Lucena (1982), Predicate Logic as a Language for Parallel Programming, *Logic Programming*, A.P.I.C. Studies in Data Processing 16, K.L. Clark and S.A. Tarnlund (ed.), Academic Press, London, pp. 189-198.

[van84] vanEmden, M.H. and Goebel, R. (1984), Waterloo Unix Prolog Tutorial Version 1.2, Logic Programming Group, University of Waterloo, Waterloo, Ontario, January.

Appendix A1
BNF Definition of IML

```
<clause>     ::= <axiom> | <query>

<query>      ::= '?' <conditions> <endclause>

<axiom>      ::= <conclusion> <imply> <conditions> <endclause>

<conclusion> ::= <literal>

<imply>      ::= <emtpy> | '<-' | ':-'

<conditions> ::= <emtpy> | <literal> <conditions>

<endclause>  ::= ';' | '.'

<literal>    ::= <atom> | <compound>

<compound>   ::= <atom> '(' <term> <termlist> ')'

<termlist>   ::= <empty> | <term> <termlist>

<term>       ::= <constant> | <variable> | <list> | <atom> | <compound>

<list>       ::= '[]' | '[' <term> <listtail> ']'

<listtail>   ::= <empty> | '|' <term> | <term> <listtail>

<constant>   ::= <atom> | <integer> | <char>
```

Appendix A2
Regular Expression Definition of IML Lexemes

```
<variable> ::= '_'(<char_set>)* | <upper>(<char_set>)*

<atom>     ::= '“'(<any_char>)*'“' | <letter>(<char_set>)*

<integer>  ::= <sign>(<digit>)+

<sign>     ::= <empty> | '-'

<char>     ::= <any char within single quotes>

<char_set> ::= <any char except 'reserve'>

<letter>   ::= <upper> | <lower>

<reserve>  ::= '(' | ')' | '[' | ']' | ';' | '?' | '|'
```

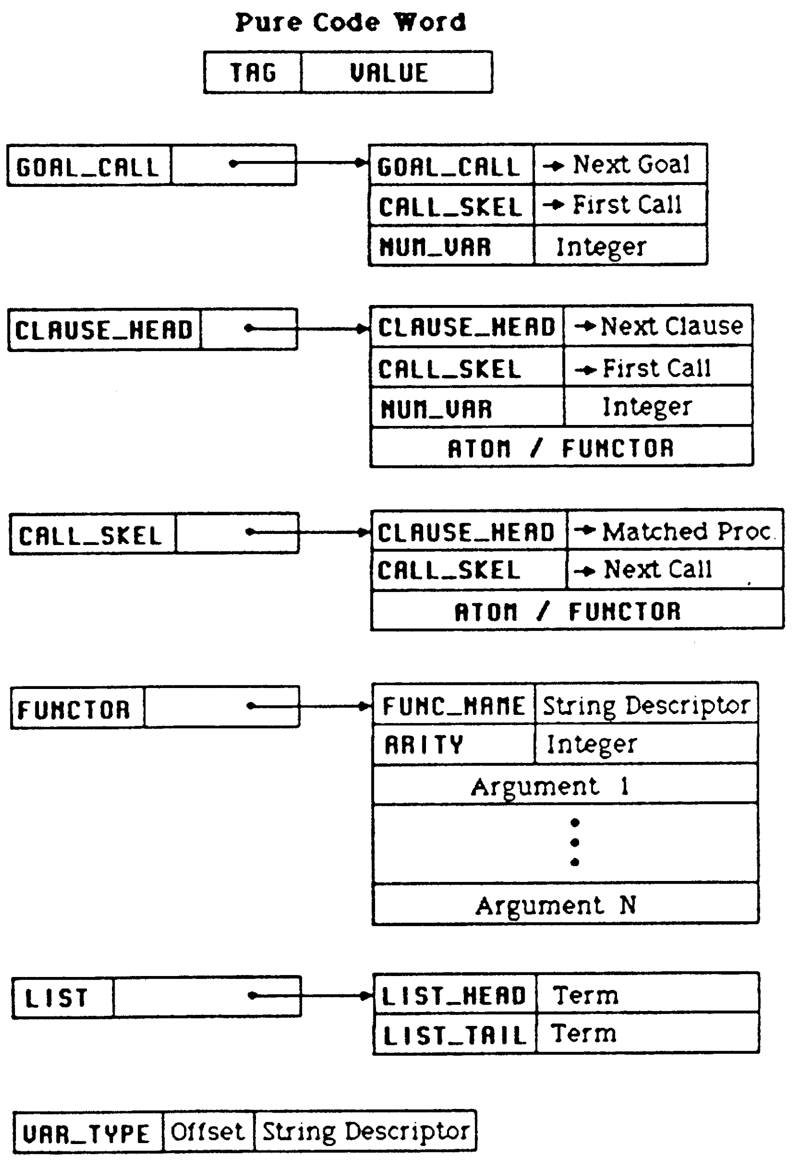
Note:
  “ “ = empty string.
  When using '“' inside a string, double it.
  When using ' as character, triple it. (i.e. ''')
  Special characters:

'\b'	backspace
'\f'	formfeed
'\n'	newline
'\t'	tab
'\ddd'	'ddd' is the decimal representation of any ascii characters.

Appendix B
BNF Definition of PIC

```
<clause>    ::= <assertion> | <query>

<query>     ::= GOAL <proc_info> <proc_call> END_PROC

<assertion> ::= PROC <proc_info> <literal> <proc_call> END_PROC

<proc_info> ::= <num_call> <num_var>

<proc_call> ::= <empty> | CALL <literal> END_CALL <proc_call>

<literal>   ::= <atom> | <functor>

<functor>   ::= <func_info> <term> <term_list> END_FUNC

<func_info> ::= <func_type> <num_arg> <name>

<func_type> ::= CFUNC | VFUNC

<term>      ::= <variable> | <constant> | <functor> | <list_unit>

<term_list> ::= <emtpy> | <term> <term_list>

<list_unit> ::= NULL_LIST | <list_info> <term> <term_list> END_LIST

<list_info> ::= <list_type> <num_term> <num_var>

<list_type> ::= CLIST | VLIST

<variable>  ::= NULL_VAR | <var_type> <offset> <name>

<var_type>  ::= BASE_VAR | REF_VAR

<constant>  ::= <atom> | <integer> | <char>

<atom>      ::= ATOM_CONST <name>

<integer>   ::= INT_CONST <value>

<char>      ::= CHAR_CONST <char>
```

Appendix C
Pure Code Representation (PC)



Appendix D
Support Routines for the ABC Algorithm

procedure	push-stack-frame (M: ModulePtr; Cl: Clause; T: NodeType)
return:	(F: FramePtr; E: EnvPtr)
effect:	allocate a new stack frame of type 'T', initialise a new binding environment for the clause 'Cl' with 'undefined' values, and return the pointers to this new frame and environment.
error:	runtime stack overflows.
procedure	pop-stack-node (F: FramePtr)
return:	none
effect:	deallocate the stack frame at location 'F' but do not erase its content.
procedure	env-ptr-of(F: FramePtr)
return:	(E: EnvPtr)
effect:	return the environment pointer of the stack frame 'F', which is based on the node type of 'F'.
procedure	is-det-node (F: FramePtr)
return:	Boolean
effect:	return true if the stack frame at 'F' is a deterministic node, otherwise return false.
procedure	set-not-real-father (F: FramePtr)
return:	none
effect:	record that the stack frame at 'F' is not the "real" father of its environment. its environment, otherwise return false.
procedure	first-call-of (Cl: Clause)
return:	(C: GoalCall)
effect:	return the first call of the clause 'Cl' if it exists, othewise return nil.
procedure	next-call-of (C: GoalCall)
return:	(C': GoalCall)
effect:	return the next call following (or to the right of) 'C' if it exists, otherwise return nil.

procedure	head-predicate-of (Cl: Clause)
return:	(H: Term)
effect:	return the head functor of the clause 'Cl'.

procedure	set-binding-limit (E: EnvPtr; C: CopyPtr)
return:	none
effect:	set the upper limits on the runtime and copy stack such that any future variable bindings above these limits are not recorded.

procedure	undo-binding (T: TrailPtr)
return:	none
effect:	starting from the top of the trail stack down to 'T' inclusive reset all the variable bindings recorded back to 'undefined'.

procedure	pop-copy-stack (C: CopyPtr)
return:	none
effect:	deallocate the storage from top of the copy stack to 'C' back to the copy stack.

procedure	swap-environment (E1,E2: EnvPtr)
return:	none
effect:	the contents of the environments E1 and E2 are swapped, and all the pointers are properly readjusted.

procedure	env-size-of (E: EnvPtr)
return:	(S: Integer)
effect:	return the size of the environment at 'E'.

Appendix E
Sample Test Programs

(1) fibonacci

```
test <- fib( 18, N ), write( N ) ;

fib( 0, 1 ) ;
fib( N, V ) <-
   seq-plus( [0,1|F], [1|F], F ),
   nth( N, F, V )
   ;
nth( 1, [A|L], A ) ;
nth( N, [A|L], V ) <-
   add( 1, N1, N ),
   nth( N1, L, V )
   ;
nth( N, seq(X,Y), V ) <-
   nth( N, X, V1 ),
   nth( N, Y, V2 ),
   add( V1, V2, V )
   ;
seq-plus( X, Y, seq(X,Y) ) ;
```

(2) naive reverse

```
test <-
   nreverse( [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20], L ),
   write( L )
   ;

nreverse( [], [] ) ;
nreverse( [X|L1], L3 ) <-
   nreverse( L1, L2 ),
   append( L2, [X], L3 )
   ;
append( [], L, L ) ;
append( [U|X], Y, [U|Z] ) <-
   append( X, Y, Z )
   ;
```

(3) quicksort

```
test <-
   qsort( [27,74,17,33,94,18,46,83,65,2,32,53,28,85,99,47,28,82,6,11], L, [] ),
   write( L )
   ;

qsort( [], R, R ) ;
qsort( [X|L], R, R0 ) <-
   partition( L, X, L1, L2 ),
   qsort( L2, R1, R0 ),
   qsort( L1, R, [X|R1] )
   ;
partition( [], _, [], [] ) ;
partition( [X|L], Y, [X|L1], L2 ) <-
   lt( X, Y ),
   partition( L, Y, L1, L2 )
   ;
partition( [X|L], Y, L1, [X|L2] ) <-
   partition( L, Y, L1, L2 )
   ;
```

(4) & (5) tower of hanoi

```
hanoi( N ) <-
   move( N, l, c, r )
   ;
move( 0, _, _, _ ) ;
move( N, L, C, R ) <-
   add( 1, N1, N ),
   move( N1, L, C, R ),
   move( N1, C, R, L )
   ;
```