THE THEORETICAL FRAMEWORK AND IMPLEMENTATION OF A PROLOG INTERPRETER

Mario Camilleri B.Ed.(Hons.)

A Dissertation in
the Department of Computing

Presented in Part Fulfilment
of the Requirements for the
Degree of Bachelor of Science
(Mathematics, Logic and Computing)
at the

UNIVERSITY
OF
MALTA

JUNE 1990

ABSTRACT

The present work investigates the theoretical framework and the interpretation algorithm of the Prolog programming language. The implementation of a structure-sharing interpreter for a subset of the language written in Modula-2 is described.

Chapter 1 introduces the concepts underlying resolution deduction procedures in general, and the Prolog programming language in particular. The equivalence of the satisfiability and deduction problems is demonstrated. The resolution principle is introduced as a refutation/deduction mechanism for propositional formulae in clausal form, and a completeness proof using semantic trees is given. It is then shown how restricting formulae to Horn clauses results in a more efficient resolution procedure called SLD-resolution. The completeness of SLD-resolution for Horn clause logic programs is demonstrated. Finally, it is shown how the addition of a search strategy to SLD-resolution results in an automatic refutation procedure. An implementation of such a procedure in Modula-2 is described.

Chapter 2 extends these refutation methods to 1st order logic. It is shown that a set S of 1st-order clauses is unsatisfiable iff it is false under all interpretations over the Herbrand universe. Moreover, S is unsatisfiable iff a FINITE set of ground instances of clauses in S is unsatisfiable, iff S has a (finite) failure tree. The resolution procedure given for propositional clauses can be LIFTED to non-ground clauses by unification. This result is used to demonstrate the completeness of resolution refutations in the 1st order case using a proof similar to the one used for the propositional case.

Chapter 3 outlines the implementation of Prolog. Following a brief review of the Prolog language, the interpretation strategy is discussed. The representation of terms constructed during unification in structure-sharing and non-structure-sharing systems is compared, and a basic optimization technique which exploits determinism in a Prolog program (deterministic-frame optimization, DFO) is outlined.

Chapter 4 describes the implementation of a small structure-sharing interpreter for a subset of the Prolog language. The interpreter is only meant to demonstrate some implementation principles and to serve as a test-bed for optimization techniques, although the design is sufficiently open to form the kernel of a more practical implementation.

June 1990

ACKNOWLEDGEMENTS

I would like to express my gratitude to the following:

My tutors, Dr V. Nezval
and
Mr V. Riolo.

Prof A. Leone Ganado, course co-ordinator.

The Department of Artificial Intelligence at the University of Edinburgh, in particular their documentation secretary Margaret E. Pithie.

TABLE OF CONTENTS

1 PROPOSITIONAL LOGIC

1.1 Objectives

This chapter introduces, within the context of propositional logic, the concepts underlying resolution deduction procedures in general, and the Prolog programming language in particular.

After a brief overview of the syntax and semantics of propositional logic, the equivalence of the satisfiability and deduction problems is demonstrated. The resolution principle is introduced as a refutation/deduction mechanism for formulae in clausal form, and a completeness proof using semantic trees is given. It is then shown how restricting formulae to Horn clauses results in a more efficient resolution procedure called SLD-resolution, which is the main computational mechanism of Prolog. The completeness of SLD-resolution for Horn clause logic programs is demonstrated. Finally, it is shown how the addition of a search strategy to SLD-resolution results in an automatic refutation procedure. An implementation of such a procedure in Modula-2 is described briefly.

1.2 Syntax

The alphabet of propositional calculus

1. a countable set *PS* of **PROPOSITIONAL SYMBOLS** (or **ATOMS**): $\{p_i \mid i \geq 0\}$
2. the set of logical connectives *LCON* : $\{\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg\}$
3. the set *AUX* of auxiliary symbols: {(,)}

Informally, the set *WFF* of well-formed (propositional) formulae is defined as the set of strings over the alphabet $AS \cup LCON \cup AUX$, such that

1. every atom is in *WFF*
2. if X is in the set *WFF,* then so is $\neg$X
3. if X and Y are in *WFF,* then so are (X$\vee$Y), (X$\wedge$Y), (X$\Rightarrow$Y) and (X$\Leftrightarrow$Y).
4. only strings generated from the application of rules 1-3 are in the set *WFF*

For convenience, we adopt the following conventions:

1. lower case letters a..z represent atoms
2. upper case letters A..Z represent arbitrary well-formed formulae (or formulae for short).
2. the brackets [and] will sometimes be used for clarity in place of (and) respectively.
4. X $\equiv$ Y will denote the logical equivalence of formulae X and Y. Note that X $\equiv$ Y is not a formula but a statement about the two formulae X and Y.
5. We sometimes drop brackets from formulae where this causes no ambiguity. In particular, outer brackets.

1.2:a DEFN: (LITERAL). A LITERAL L is an atom p (a positive literal) or its negation $\neg$p (a negative literal).

1.3 Semantics

We start by defining the set BOOLEAN = {FALSE,TRUE}, where FALSE and TRUE are called TRUTH VALUES, and a function CONJUGATE : BOOLEAN $\rightarrow$ BOOLEAN defined as:

CONJUGATE(TRUE) = FALSE
CONJUGATE(FALSE) = TRUE

1.3:a DEFN: *(BASE).*

The BASE Bs of a finite set S of formulae is the (finite) set {p1, p2,..,pn} of atoms pi $\in$ PS which appear in S.

1.3:b DEFN: *(TRUTH ASSIGNMENT).*

A **TRUTH ASSIGNMENT** τ for a set of formulae S is a function $\tau:B_s \rightarrow$ *BOOLEAN,* We write $\tau(p)$=**TRUE** to denote that τ assigns the truth-value **TRUE** to an atom p.

1.3:c DEFN: *(INTERPRETATION).*

An **INTERPRETATION** I for a set of formulae S is a function $I:S \rightarrow$ *BOOLEAN,* defined as follows:

1. for any atom p, $I(p) = \tau(p)$
2. for any non-atomic formula $\neg A$, $I(\neg A) = CONJUGATE(I(A))$
3. for any non-atomic formula A, and formulae X and Y, $A=X*Y$ ($* \in LCON \setminus \{\neg\}$), I(A) is a function of I(X),I(Y) and $*$, as follows:

I(X)	I(Y)	$I(X \vee Y)$	$I(X \wedge Y)$	$I(X \Rightarrow Y)$	$I(X \Leftrightarrow Y)$
T	**T**	T	T	T	T
T	**F**	T	F	F	F
F	**T**	T	F	T	F
F	**F**	F	F	T	T

where **F** and **T** stand for the truth values **FALSE** and **TRUE** respectively. We write I(A)=TRUE to denote that the extended interpretation I assigns the value TRUE to the formula A. Clearly, I is a compositional extension of the truth-assignment τ, and the term **INTERPRETATION** will henceforth denote both.

An interpretation I induces a partition of B_s into two sets $B_{sf}=\{p \in B_s : I(p)=FALSE\}$, and $B_{st}=\{p \in B_s : I(p)=TRUE\}$. Conversely, for every bipartition of B_s there corresponds a unique interpretation. It is thus convenient to represent an interpretation as a set of literals $I=\{L_1,..,L_n\}$ such that $L_i=p_i$ if $p_i \in B_{st}$, and $L_i=\neg p_i$ if $p_i \in B_{sf}$.

Since there are 2^n bipartitions of B_s (where $n = |B_s|$), it follows that there are 2^n possible distinct interpretations of S.

1.4 Satisfiability

A interpretation I is said to **SATISFY** a formula A if I(A)=TRUE. We denote this by $I \vDash A$. I is then called a **MODEL** of A. The formula A is said to be **SATISFIABLE** (or **CONSISTENT**) if there exists an interpretation which is a model of A. Otherwise, A is said to be **UNSATISFIABLE.** An interpretation I cannot satisfy both A and $\neg A$, because $I(\neg A)=CONJUGATE(I(A))$. Hence $(A \wedge \neg A)$ is an unsatisfiable formula.

A is said to be a **TAUTOLOGY** (or **VALID**) if every interpretation of A is also a model of A, and we denote this by $\vDash A$. Hence, A is a tautology iff $\neg A$ is unsatisfiable.

Two formulae A and B are said to be **LOGICALLY EQUIVALENT,** denoted $A \equiv B$, if $\vDash A \Leftrightarrow B$, ie if I(A)=I(B) for all interpretations I.

An interpretation I is said to satisfy a set of formulae $\Gamma=\{ß_1,..,ß_n\}$ iff $\forall ß_i \in \Gamma$, $I \vDash ß_i$. Thus a set of formulae is satisfiable (or **CONSISTENT**) if all its members admit a common model. In this sense, $\Gamma \equiv ß_1 \wedge .. \wedge ß_n$, ie Γ can be viewed as a conjunction of the formulae $ß_i$.

1.5 Clausal Form And Conjunctive Normal Form

1.5:a DEFN: *(CLAUSE)*.

A **CLAUSE** is a disjunction of a finite number of literals: $(L_1 \vee .. \vee L_n)$ $n \geq 0$. When n = 0, the clause is called an **EMPTY** (or **NULL**) clause, denoted by □. When n=1 the clause consists of a single literal (L) and is called a **UNIT CLAUSE**.

1.5:b DEFN: *(CONJUNCTIVE NORMAL FORM)*.

A conjunctive normal form (CNF) is a conjunction of a finite number of clauses: $(C_1 \wedge .. \wedge C_m)$.

1.5:c THEOREM:

For every wff A in the propositional calculus there exists a formula A' in CNF such that A≡A'.

A constructive proof may be found in any textbook on logic, for example [DOW86 pp22ff],[THA88 pp15ff].

Henceforth assume that all formulae are in CNF.

A clause can be represented as a set of literals $\{L_i \mid i = 1,..,n\}$. Similarly, a CNF can be represented as a set of clauses $\{C_j \mid j = 1,..,m\}$, each Cj being a clause $\{L_{ji} \mid i = 1,..,n\}$.

For convenience the set-representation of clauses and conjunctive normal forms will be used henceforth. Thus, the conjunctive formula

(A∨B) ∧ (C∨¬D) ∧ (E)

will be represented as the set of clauses

{{A,B} , {C,¬D} , {E}}.

The semantics of clauses and CNFs follow from the more general semantics of WFFs. For a clause $C = \{L_1,..,L_n\}$:

a. I(C)=TRUE iff $\exists L_i \in C$ such that $I(L_i)$=TRUE.
b. Hence the empty clause □ is unsatisfiable - ie I(□)=FALSE for all possible interpretations I, including the null interpretation I={}. □ is the only clause falsified by the null interpretation.
c. $\models C \Leftrightarrow \exists L_i, L_j \in C$ such that $L_i = \neg L_j$

For a set of clauses (CNF) $S = \{C_1,..,C_n\}$:

a. I(S)=TRUE iff $\forall C_i \in S$, I(C)=TRUE.
b. Since □ is unsatisfiable, a set of clauses which contains □ as a member is unsatisfiable.

1.6 Logical Consequence

A formula Δ is said to be a **LOGICAL CONSEQUENCE** of a set of formulae $\Gamma=\{ß_1,..,ß_n\}$ if every interpretation which satisfies Γ also satisfies Δ, denoted $\Gamma \models \Delta$. If Γ is unsatisfiable, then $\Gamma \models \Delta$ for all formulae Δ.

1.6:a LEMMA:

If $\Gamma \models \Delta$ and $\Gamma \models \Omega$, then $\Gamma \models (\Delta \wedge \Omega)$.

PROOF: since all interpretations which satisfy Γ also satisfy Δ, and all interpretations which satisfy Γ also satisfy Ω, all interpretations which satisfy Γ also satisfy $(\Delta \wedge \Omega)$. Hence $\Gamma \models (\Delta \wedge \Omega)$.

1.6:b COROLLARY:

$\Gamma \models \Delta$ and $\Gamma \models \neg\Delta$ iff Γ is unsatisfiable.

PROOF: by lemma 1.6:a, $\Gamma \models \Delta$ and $\Gamma \models \neg\Delta$ implies that $\Gamma \models (\Delta \wedge \neg\Delta)$. But since no interpretation satisfies $(\Delta \wedge \neg\Delta)$, then no interpretation satisfies Γ. Hence Γ is unsatisfiable.

1.6:c THEOREM: *(DEDUCTION PRINCIPLE)*
$\Gamma \models \Delta$ if and only if $\Gamma \cup \{\neg\Delta\}$ is unsatisfiable.

PROOF:

($\Rightarrow$) $\Gamma \models \Delta \Rightarrow \Gamma \cup \{\neg\Delta\}$ is unsatisfiable.

For any interpretation I, either I(ß)=TRUE for all $ß \in \Gamma$ and I(Δ)=TRUE (hence I(¬Δ)=FALSE), or I(ß)=FALSE for some $ß \in \Gamma$. Either way, $I(\Gamma \cup \{\neg\Delta\})$=FALSE.

($\Leftarrow$) $\Gamma \cup \{\neg\Delta\}$ is unsatisfiable $\Rightarrow \Gamma \models \Delta$.

For any interpretation I, either I(ß)=TRUE for all $ß \in \Gamma$ and I(¬Δ)=FALSE (hence I(Δ)=TRUE), or I(ß)=FALSE for some $ß \in \Gamma$. Thus I(Δ)=TRUE whenever I(Γ)=TRUE, and so $\Gamma \models \Delta$.

1.7 Deduction Procedures

The deduction principle thus reduces the problem of inferring a goal formula A from the hypothesis set Γ to the equivalent problem of determining whether the formula set $\Gamma \cup \{\neg A\}$ is satisfiable or not. Since much of the application of logic in mainstream AI (including Prolog) involves automatic deduction, the development of efficient algorithms for determining the satisfiability of a set of formulae is clearly of paramount importance.

[CHA74] presents a thorough treatment of many of the seminal work in the field of automated theorem proving, such as that of Davis and Putnam. A review of more recent developments in the field may be found in [LOV84].

In the context of Prolog, the most important developments were in the area of resolution deduction. An extension of a theorem by Herbrand and of the work of Davis and Putnam, the resolution principle was introduced by Robinson in 1965 [ROB65]. The algorithm was further refined in subsequent years by Robinson, Kowalski and others [CHA74][GAL87] (in particular by restricting its use to the class of Horn formulae). Kowalski's work led directly to the development of Prolog by Colmerauer and Roussel at Marseille in the early 1970s.

In this chapter, the application of resolution to the propositional calculus will be discussed, together with a few refinements to the basic algorithm. At the end of this section, an interpreter for a propositional version of Prolog will be developed to illustrate implementation techniques. In the next chapter, the resolution principle will be extended to the 1st order case.

1.8 Semantic Trees

DEFN: *(SEMANTIC TREE)*.
Let $S=\{C_1,..,C_n\}$ be a set of clauses and $B_s=\{p_1,..,p_m\}$ its base. A **SEMANTIC TREE** T_s is a complete binary tree with m levels (with the root at level 1) such that all left/right edges emanating from nodes at level i are labelled $\neg p_i/p_i$.

EXAMPLE:

S = { {p,¬q} , {q,r} , {¬p,¬r} }
B_s = { p, q, r }
T_s =

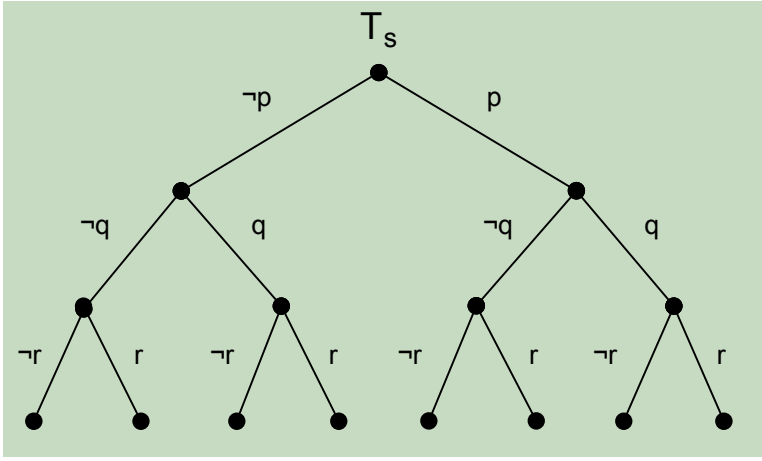


Figure 1: Semantic Tree

Let $I_N=\{L_1,..,L_n\}$, where L_i is either p_i or $\neg p_i$, be the path from the root of a semantic tree T_s to node N. Then I_N corresponds to a (partial) interpretation for S. Similarly, every root-to-leaf path in T_s corresponds to an interpretation of S, and vice-versa.

DEFN: (FAILURE NODE).

A node N in a semantic tree is said to be a **FAILURE NODE** if I_N falsifies some clause in S but I_M satisfies S for every ancestor node M of N. If a set of clauses S is unsatisfiable, then every path in the semantic tree T_s must pass through a failure node.

DEFN: *(FAILURE TREE)*.

For a set of clauses S, let T' be the tree obtained from the semantic tree T_s by truncating all paths at failure nodes. If all leaves of T' are failure nodes for S, then T' is called a **FAILURE TREE** and denoted FT_s. Each leaf of FT_s is labelled with the set of clauses falsified by the corresponding interpretation.

EXAMPLE:

Let S = { {a,b} , {a,¬b} , {¬a} }

Then the following is a failure tree of S

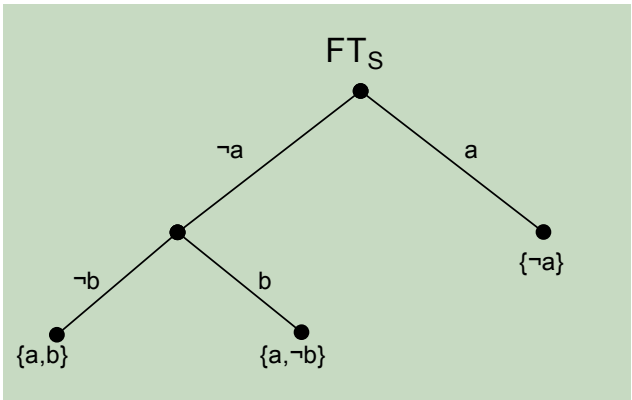


Figure 2: Failure tree

1.8:d LEMMA: A clause set S has a failure tree if and only if S is unsatisfiable.

PROOF:

(⇒) Let FT_s be a failure tree for S. By definition, every path in FT_s terminates in a failure node. Hence every interpretation of S falsifies some clause in S. Hence S is unsatisfiable.

(⇐) Let S be unsatisfiable. Then every interpretation I falsifies some clause in S. Hence every path in the semantic tree T_s terminates in a failure node - and therefore S has a failure tree.

A refutation of S is a traversal of the semantic tree T_s in search of the failure tree FT_s.

1.8:e LEMMA:

A set of clauses S has a failure tree of one node iff S contains the empty clause □.

PROOF:

The failure tree of one node corresponds to the null interpretation I={}. Since □ is the only clause falsified by the null interpretation, it follows that if S is falsified by the null interpretation it must contain the empty clause. Conversely, if S contains □, then S is falsified by the null interpretation, and therefore its failure tree must be the one-node tree.

1.8:f DEFN: *(INFERENCE NODE)*.

A node N in a failure tree is said to be an **INFERENCE NODE** if BOTH child nodes of N are failure nodes (and therefore leaves of FT_s).

1.8:g LEMMA:

Every failure tree (with the exception of the trivial one-node tree) must have at least one inference node.

PROOF:

Assume FT_s is a failure tree of more than one node which has no inference nodes. Then every node has at least one non-failure descendant, and therefore we could find a root-to-leaf path in FT_s without failure nodes, corresponding to an interpretation which satisfies S. But this is a contradiction, since S is unsatisfiable.

1.9 Resolution

In the propositional case, the resolution inference rule is essentially Gentzen's cut rule [SMU68], itself a generalization of modus ponens.

1.9:a LEMMA:

$[(A \vee P) \wedge (B \vee \neg P)] \equiv [(A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)]$ for all formulae A,B and P.

PROOF:

a. $[(A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)] \Rightarrow [(A \vee P) \wedge (B \vee \neg P)]$

If an interpretation $I \vDash [(A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)]$ then by definition $I \vDash (A \vee P)$ and $I \vDash (B \vee \neg P)$. Thus $I \vDash [(A \vee P) \wedge (B \vee \neg P)]$

b. $[(A \vee P) \wedge (B \vee \neg P)] \Rightarrow [(A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)]$

This implication is of the form

$X \Rightarrow X \wedge Y$
$\rightarrow \neg X \vee (X \wedge Y)$
$\rightarrow (\neg X \vee X) \wedge (\neg X \vee Y)$
$\rightarrow (\neg X \vee Y)$
$\rightarrow X \Rightarrow Y$

Hence, it suffices to show that

$[(A \vee P) \wedge (B \vee \neg P)] \Rightarrow (A \vee B)]$

This eventually simplifies to the CNF

$(A \vee B \vee \neg A \vee \neg B) \wedge (A \vee B \vee \neg B \vee \neg P) \wedge (A \vee B \vee \neg A \vee P) \wedge (A \vee B \vee P \vee \neg P)$

Which is a tautology, since each disjunction contains at least one conjugate pair of formulae.

In particular, (1) holds for arbitrary disjunctions A,B and atomic formula P. Hence, the set of clauses

{{A,P} , {B,¬P}}

is logically equivalent to

{{A,P} , {B,¬P} , {A,B}}

ie, we can add a new clause, {A,B}, without affecting the satisfiability of the set. In general, the set

{C_1, .., C_n, {A,P}, {B,¬P}, {A,B}}

is unsatisfiable iff

{C_1, .., C_n, {A,P}, {B,¬P}}

is unsatisfiable [GAL87 p128].

The clause {A,B} is called the **RESOLVENT** of the clauses {A,P} and {B,¬P}, which are called **PARENT CLAUSES**, and the process of adding a resolvent of two parent clauses from a set to that set is called a **RESOLUTION STEP**.

1.9:b DEFN: *(RESOLVENT)*.
The **RESOLVENT** of two parent clauses C_1 and C_2 with respect to some literal L, $L \in C_1$ and $\neg L \in C_2$, is a clause C_R such that $\mathbf{C_R = (C_1 \backslash \{L\}) \cup (C_2 \backslash \{\neg L\})}$

In particular, if $C_1 = \{L\}$ and $C_2 = \{\neg L\}$, then $C_R = \{\}$ - ie the resolvent of the two clauses {P} and {¬P} is the empty clause □. A set of clauses which has □ as a resolvent is unsatisfiable as seen above.

1.9:c DEFN: *(RESOLUTION CLOSURE)*.
The resolution closure S^* of a set of clauses S is the closure of the set S under the operation of resolution.

1.10 The Method Of Refutation By Resolution

Starting with some set of clauses, S, the method of refutation by resolution attempts to derive the resolvent □ by successive application of resolution steps:

$S \rightarrow \{S, R_1,..,R_n, \square\}$.

The procedure can be stated as

```
WHILE (□ ∉ S) DO
BEGIN
  Select literal L and clauses C1,C2 ∈ S such that L∈C1, ¬L∈C2
  compute their resolvent R = C1\{L} ∪ C2\{¬L}
  add the clause R to S
END
```

EXAMPLE:

Consider a refutation of the set

S = {{a,¬b},{c,b},{¬c},{c,¬a}}

1. {a,¬b}
2. {c,b}
3. {¬c}
4. {c,¬a}

5. {a,c} /* Res(1,2) */
6. {¬a} /* Res(3,4) */
7. {a} /* Res(3,5} */
8. □ /* Res(6,7) */

EXAMPLE:

Consider a deduction of $\Delta=\{a,\neg b,c\}$ from the hypothesis set $\Gamma=\{\{a,d\},\{c,\neg d\},\{\neg c\}\}$.

By the deduction principle (Theorem 1.6:c), $\Gamma \models \Delta$ iff $\Gamma \cup \neg\Delta$ is unsatisfiable.

We convert $\neg\Delta$ to CNF, giving $\{\{\neg a\},\{b\},\{\neg c\}\}$, and then use the resolution procedure to try and refute the set $\{\{a,d\},\{c,\neg d\},\{\neg c\},\{\neg a\},\{b\}\}$

1. {a,d}
2. {c,¬d}
3. {¬c}
4. {¬a}
5. {b}
6. {a,c} /* Res(1,2) */
7. {a} /* Res(6,3) */
8. □ /* Res(7,4) */

Since the set $\Gamma \cup \neg\Delta$ is unsatisfiable, $\Gamma \models \Delta$.

1.11 Completeness Of The Resolution Principle (Propositional Case)

To prove the completeness of resolution refutation we first demonstrate the relationship between resolution and failure trees.

Consider an inference node i in a failure tree FT_s of an unsatisfiable set of clauses S, whose child nodes j and k falsify the two clauses C_j and C_k.

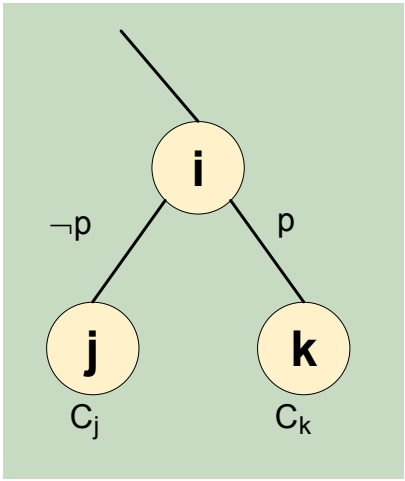


Figure 3: Inference Node

Since C_j fails at node j, then it must contain the literal p. Similarly, C_k must contain the literal ¬p. Hence, the two clauses resolve on p to produce the resolvent clause $C_R = C_j\backslash\{p\} \cup C_k\backslash\{\neg p\}$. We note that

a. Since C_j fails at node j, then all the literals in $C_j\backslash\{p\}$ must be false at or above node i, and similarly
b. Since C_k fails at node k, then all the literals in $C_k\backslash\{\neg p\}$ must be false at or above node i.

Hence C_R must fail at or above node i. Consequently, the set $S \cup C_R$ has a smaller failure tree than the set S.

1.11:a THEOREM: *(COMPLETENESS OF THE RESOLUTION PRINCIPLE - PROPOSITIONAL CASE)*.

A (finite) set of clauses S is unsatisfiable iff the empty clause □ can be deduced from S by resolution.

PROOF:

(⇒) Proof by induction on the number of nodes in FT_s, the failure tree of S. Let S be unsatisfiable. Then S has a failure tree FT_s. If FT_s has only a single node, then S must contain the empty clause □, and so the theorem is proved.

So assume FT_s has more than one node. Then it must have at least one inference node (by lemma 1.8:g), say node i. Let j and k be the failure nodes immediately below i, and let C_j and C_k be the clauses in S which are falsified by the partial interpretations I_j and I_k. As shown above, the resolvent C_R of the clauses C_j and C_k must fail at or above node i, and hence the failure tree FT_s' of the set $S'=S \cup C_R$ must have at least two fewer nodes than FT_s. Hence by induction, resolution must eventually derive the one-node failure tree, for either FT_s' contains only a single node, or FT_s' contains at least one inference node.

(⇐) Assume S is satisfiable, but that there is a deduction of □ from S. Thus the resolution closure S^* of S contains □, and is therefore unsatisfiable. Which is a contradiction, since resolution preserves satisfiability (by lemma 1.9:a)

1.12 Refutation Trees

A resolution derivation of a clause from the clause set S can be represented as an **ORIENTED BINARY TREE** [KNU73 pp372ff], called a **RESOLUTION TREE**, in which the leaves are labelled with clauses from the set S, and each internal node is labelled with the resolvent of its children nodes. If the root of the tree is labelled with □, then the tree is called a **REFUTATION TREE**.

For example, a refutation of the set

S = {{a,¬b},{c,b},{¬c},{c,¬a}}

can be represented by the refutation tree

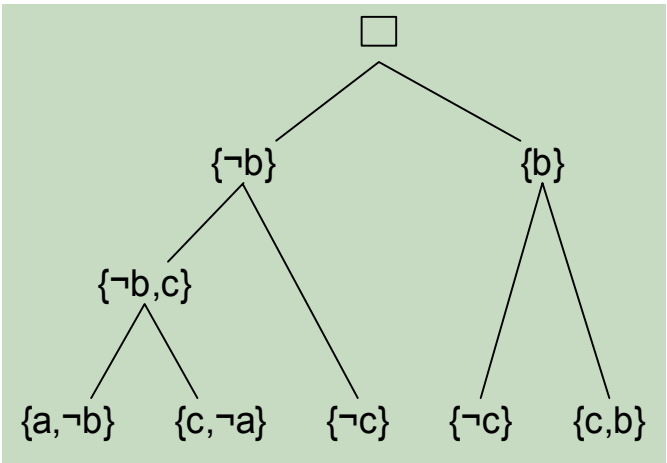


Figure 4: Refutation Tree

The resolution procedure attempts to construct a refutation tree from the bottom up. The algorithm is non-deterministic since in general there is more than one choice for C_1, C_2 and L. The strategy adopted in selecting which clauses and literals to resolve is crucial to the efficiency of the algorithm. Ideally, given an unsatisfiable set of clauses S, the procedure should construct the refutation tree with the smallest number of nodes. However, the problem is known to be NP-complete for the class of general propositional formulae [DOW84].

Various refinements to the basic resolution procedure have been proposed with the aim of improving its efficiency, sometimes at the cost of completeness [CHA74][LUK70]. Refinement theorems attempt to restrict the search space by limiting choice in selecting parent clauses and/or the literal to resolve upon. One such refinement restricts the algorithm to the Horn-Clause subset of propositional formulae, making it possible to develop satisfiability tests that run in polynomial time.

1.13 Horn Clauses

A **HORN CLAUSE** is a clause with at most one positive literal. We will represent the Horn clause

{p,¬b,¬c,¬d} by **p :- b,c,d.**

p is called the **HEAD** of the clause, and **b,c,d** the **BODY.** Note that the body of a clause is a **CONJUNCTION** of literals. A Horn clause which contains a positive literal is called a **DEFINITE HORN CLAUSE**. A Horn clause consisting solely of negative literals is called a **NEGATIVE HORN CLAUSE**. A definite clause which consists of a single (positive) literal is called a **UNIT** clause, and is written **p.** .

A unit clause **p.** is TRUE if and only if **p** is TRUE. If **b,c** and **d** are TRUE, then the Horn clause **p :- b,c,d.** is TRUE if and only if **p** is also TRUE.

Non-unit definite Horn clauses model **RULES**, the body representing a set of premises with the head as conclusion. A unit clause, which has a null body, models a **FACT**. A **HORN-CLAUSE LOGIC PROGRAM** consists of a set of definite horn clauses (rules and facts), called the **DATABASE**, together with a **NEGATIVE** horn clause called the **GOAL**. The goal represents a negated **QUERY** formula - for example, if QUERY = $A \wedge B$, then GOAL = $\neg(A \wedge B)$ = $\{\neg A \vee \neg B\}$ = :-A,B. By the deduction theorem, QUERY is a logical consequence of the database iff DBASE $\cup$ GOAL is unsatisfiable, iff resolution can infer $\square$ from DBASE $\cup$ GOAL.

1.14 Restricting Resolution To Horn Clauses Programs - Sld-Resolution

1.14:a LEMMA:
Horn clauses are closed under resolution.

1.14:b LEMMA:
Definite Horn clauses are closed under resolution.

Because definite Horn clauses are closed under resolution, $\square$ can never be derived from a set which contains no negative clauses. In a logic program, there is exactly one negative clause N_0, the goal, together with a set of definite clauses $D_1,..,D_n$, the database.

A variant of the resolution procedure, called **SLD-RESOLUTION** (**S**elected **L**inear with **D**efinite clauses), can be used with Horn-clause logic programs to restrict the choice of clauses for resolution.

At each step of an SLD-derivation, one of the parent clauses (the **CENTRE CLAUSE**) is the resolvent of the previous step, while the second (the **INPUT CLAUSE**) is a member of the database. Of necessity, the initial centre clause is the goal. Each SLD-resolution step suppresses one literal from the centre clause (called the **SELECTED LITERAL**), and introduces 0 or more literals from the input clause. The refutation terminates successfully when the centre clause becomes empty. A derivation/refutation using SLD-resolution is called an **SLD-DERIVATION/REFUTATION**.

An SLD-refutation has a linear refutation tree:

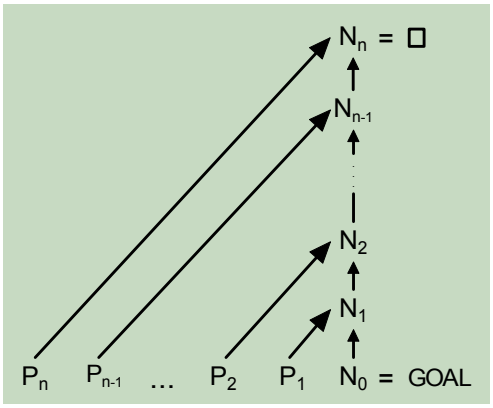


Figure 5: SLD-Refutation Tree

where the D_is are not necessarily distinct. Thus an SLD-refutation is a sequence $N_0,N_1,N_2,..,N_n$ of negative clauses, where N_0 is the goal, N_n is □, and N_i (i>0) is the resolvent of N_{i-1} with a definite clause from the database.

1.14:e THEOREM: *(COMPLETENESS OF SLD-RESOLUTION FOR HORN LOGIC ROGRAMS - PROPOSITIONAL CASE)*.

A set of propositional Horn clauses S with exactly one negative clause (a Horn program) has a resolution refutation iff it has an SLD refutation.

PROOF:

(⇐) by definition. An SLD refutation is a resolution refutation.

(⇒) We show that a refutation tree T for the Horn set S can be transformed into a linear refutation tree T' (the process is sometimes called **LINEARIZATION,** see [GAL87 pp422ff]).

a. transform T into T_1 by ordering the children of each internal node of T so that the left child contains the positive literal resolved upon, and the right the negative literal. Because the right child clause supplies the NEGATIVE literal, each internal node in T_1 is labelled with a clause having the same head as its right child clause. In particular, the right child clause of the root □ must be a negative clause, and therefore clauses labelling nodes along the rightmost path of T_1 must all be negative. Since all the leaves of T_1 are labelled with clauses from S (which has only one negative clause) its rightmost leaf must be the goal.

b. The next step is to transform T_1 into T' by making the left child of every internal node a leaf while preserving the clause labelling the root of the tree. Consider the tree T_1

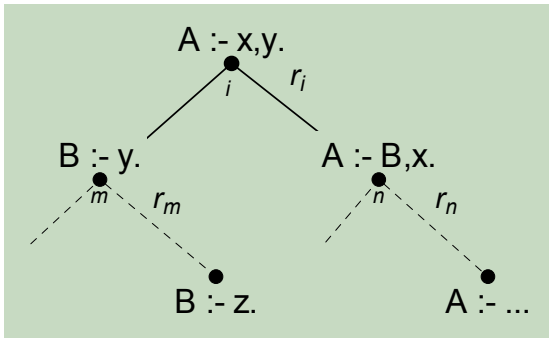


Figure 6

where A and B are propositions, and x,y and z are proposition sets. All clauses labelling the nodes along the rightmost path of T_1 have A as their head. Similarly, nodes along the rightmost path of subtree T_m have B as their head. Let node o, labelled B:-z, be the rightmost leaf of T_m. Hence the net effect of T_m is to transform B:-z (a

member of S) into B:-y by resolving on some literal in z. Replacing the whole subtree T_m by the leaf o, we get

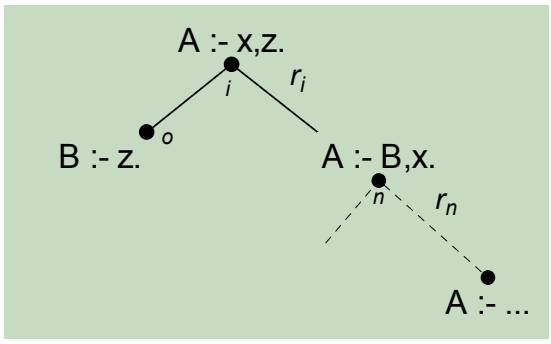


Figure 7

The new root is A:-x,z, which can be transformed into A:-x,y by using the resolution subtree T_m which changed B:-z into B:-y in the original tree:

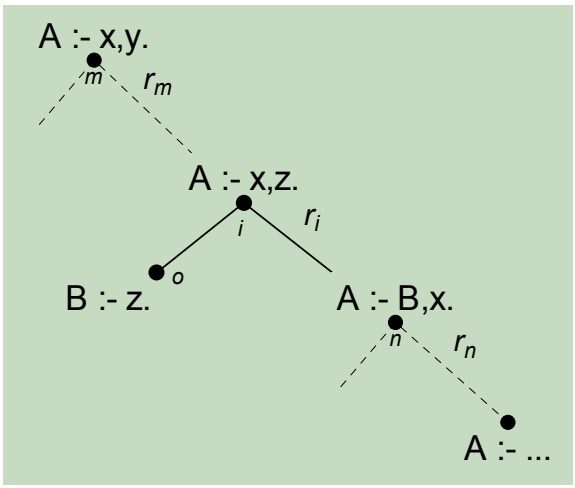


Figure 8

By repeating the algorithm at each internal node of T_1, the tree T', in which each internal node has a leaf as its left child and □ at its root (since the transformation preserves the root node), is derived. T' is a linear SLD refutation tree as defined above.

1.15 Choice Of Selected Literal In SLD-Resolution

In SLD-derivation, the following choices have to be made at each step in the process of refuting a Horn program:

a. choice of selected literal
b. choice of input parent clause in the event that multiple clauses in the database have the selected literal as their head.

The following theorem allows the adoption of a deterministic strategy in choosing the selected literal. A proof of the theorem is given in [APT82 p849].

1.15:a THEOREM:

Let $N_0,..,N_n$ (=□) be an SLD-refutation for a Horn program with goal N_0. Then, for each literal L_i of N_j, there exists an SLD-refutation in which L_i is the selected literal.

The completeness of SLD-refutation is thus independent of the choice of literal at each step. We can arrange for a scheme which selects the literal using some deterministic strategy without sacrificing completeness. The scheme used in Prolog is to order the literals in the body of a clause from left to right, and always to select the leftmost literal in the centre clause for the next

resolution step. The new centre clause is constructed by concatenating the body of the input clause to the FRONT of the old centre clause.

Although the strategy chosen fro selecting the literal is immaterial to the completeness of the SLD procedure, it may affect the size of the derivation tree in the case that the program is non-refutable. For example, if the goal were :-a,b,c,d,e. and no clause in the database has e as the head, the sooner e is chosen as the selected literal, the smaller the derivation tree that has to be constructed before the program is known to be non-refutable.

1.16 Search Strategies

The set of clauses in S having the same literal L as their head is called the **CANDIDATE SET** for L. The strategy chosen to select an input clause from the set of candidates for the selected literal is called a **SEARCH STRATEGY**.

Suppose the centre clause in a derivation is N_i = :-L,x. with L the selected literal and x the sequence of literals forming the tail of the clause, and assume that there are n candidates $\{C_1,..,C_n\}$ for L in the database. Each candidate C_j corresponds to the derivation tree with C_j as the selected input clause for N_i. The forest of all possible derivation trees for a Horn program is called the **SEARCH SPACE**.

The search space can be considered a tree (the **SEARCH TREE**), each node of which is labelled with a centre clause (or □). The root is labelled with the goal clause. Each non-empty node has one descendant for every candidate input clause for the selected literal. Each path in the search tree corresponds to a derivation. A derivation whose end-node is □ is a refutation.

For example, the search tree for the program

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

is shown in figure 9.

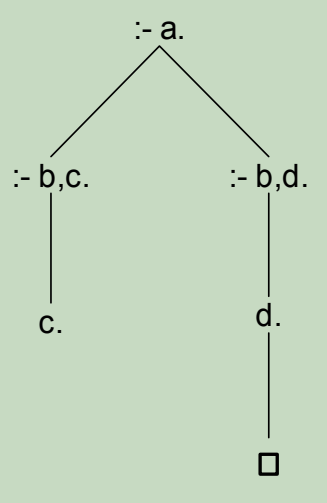


Figure 9: Search Tree

The search strategy is thus a traversal of the search tree. The search is successful (ie results in a refutation) when a node labelled with □ is encountered.

Such a traversal can be performed in either a depth-first or a breadth-first order (see, for example, [GOO85 pp139ff]). A depth-first search follows one derivation (path) down to the leaf. If the leaf is not □, then an alternative derivation is considered by backtracking to the most recent branch node and selecting an untried path. A breadth-first search traverses the search tree level by level, considering all possible derivations simultaneously until a refutation (if one

exists) is discovered. The problem with the breadth-first search is the large amount of space required to store a centre clause for each possible derivation in the search space.

The depth-first search, although more efficient to implement, is not complete. Consider the search tree for the following program:

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

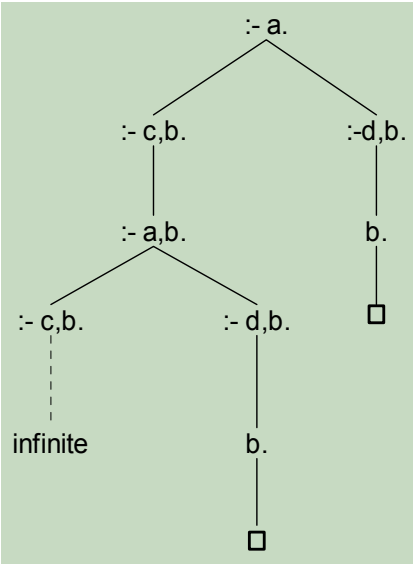


Figure 10: Depth-first Search

The program is clearly refutable, but a depth-first search which traversed the search tree in a left-to-right order would go into an infinite loop down the leftmost path and never discover the refutation in the rightmost path. Despite this, depth-first traversal is employed as the search strategy in most Prolog implementations. One of the few exceptions is Parlog86 [RIN88], which combines a depth-first search with a (guarded) breadth-first search in a parallel implementation of Prolog (see also [CON89]).

Although SLD-resolution with depth-first search is not complete, it is sound - ie SLD-resolution only derives □ from a program S if S is unsatisfiable.

1.17 SLD-Refutation With Depth-First Search

The following is an SLD-refutation procedure using a depth-first search. S is a set of definite Horn clauses, and G the goal clause consisting of a list of (negative) literals. We use FIRST(G) to mean the first literal in G, REST(G) the list G - FIRST(G). We let SELECT(L) be a procedure which, on successive calls, returns a different clause C ∈ S having L as its head, or NIL if there are no (more) such clauses.

```
PROCEDURE Satisfy (G : goal) : BOOLEAN;
BEGIN
   IF G=□ THEN
      RETURN TRUE
   END;
   WHILE (C := SELECT(FIRST(G))) ≠ NIL
      IF Satisfy(BODY(C)+REST(G)) THEN RETURN TRUE;
   END;
   RETURN FALSE;
END Satisfy;
```

1.18 Implementation

A propositional version of Prolog (called PROPOS) using SLD-resolution with depth-first search was implemented in Modula-2 for an MSDos machine (see Appendix A). The syntax of PROPOS is identical to that of Horn clauses as introduced above. The following CFG defines the syntax accepted by the PROPOS parser:

```
<program> ::= <clause> , { <clause> }
<clause>  ::= <atom> [ :- <body> ] .
<body>    ::= <atom> { , <atom> }
<atom>    ::= a..z { a..z }
<goal>    ::= :- <body> .
```

In addition, PROPOS supports the following commands, which are introduced by a period:

```
.exit             terminate execution.
.listing          list all clauses in database.
.retract <atom>   retract all clauses in the database having the
                  symbol <atom> as head.
```

The program is called from the operating system prompt with

```
PROPOS [ <file specification> ]
```

where the optional <file specification> specifies a file of clauses to be automatically loaded into the database at startup. The file may contain definite clauses, goal clauses and commands.

Database Representation

PROPOS employs a simple hashing scheme for the storage of propositional symbols (atoms), based on the first character of the symbol. The hash table is implemented as an array CLAUSES indexed by the characters 'a' .. 'z' :

```
clauses : ARRAY ['a' ..  'z'] OF HeadPtr;
```

Each entry in the array CLAUSES is a pointer to a linked list of propositional symbols starting with the corresponding character. Each record in the list is a HeadRec having the following format:

```
HeadRec  =  RECORD
            sym    : symbol name as a string (currently the maximum
                     length of a symbol name is 30 characters).
            nxt    : pointer to next propositional symbol starting
                     with the same character.
            clause : pointer to a linked list of clauses having this
                     propositional symbol as their head.
            END;
```

Each clause is stored as a linked list of AtomRecs representing the propositional symbols in the body of the clause. The format of an AtomRec is as follows:

```
AtomRec  =  RECORD
            sym    : pointer to HeadRec for this symbol.
            nxt    : pointer to next atom in the body of this
                     clause.
            END;
```

The linked list of AtomRecs representing the clause body is linked to the HeadRec representing the clause head by a BodyRec, which is declared as follows:

```
BodyRec  =  RECORD
            nxt    : pointer to next clause for this head symbol.
            first  : pointer to first AtomRec in the clause body.
            END;
```

Figure 11 shows how the clause set

```
a :- d1,d2.
d1.
d2 :- d1,g1,i1.
```

is stored in the database.

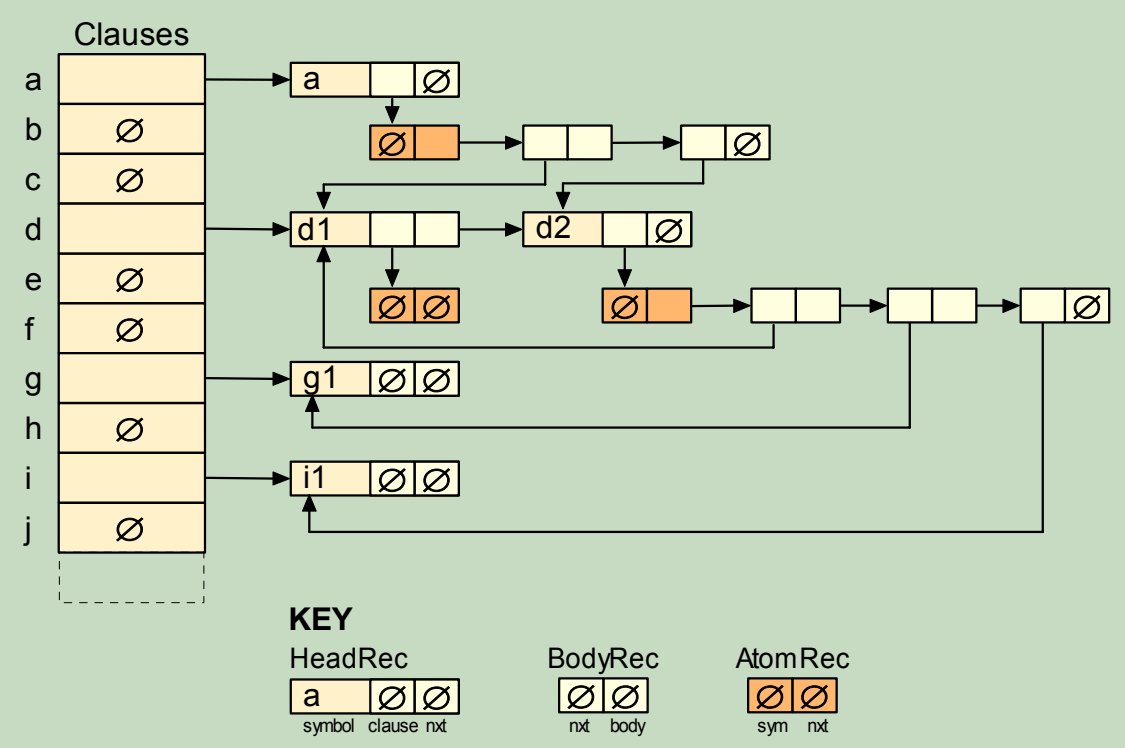


Figure 11: PROPOS database

The refutation algorithm is similar to the *Satisfy* procedure given above. The goal clause read in from the user is stored in a clause structure (ie HeadRec, BodyRec and AtomRecs) pointed to by variable *goal*. Procedure *Prove* is called with *goal* as parameter, and attempts to satisfy each of the literals in the body of the goal clause. The algorithm is given below in pseudo code.

```
PROCEDURE Prove (goal : Ptr to a HeadRec);
   PROCEDURE ProveClause(goal : Ptr to a HeadRec) : BOOLEAN;
   VAR b : Ptr to a BodyRec;
       t : BOOLEAN;

       PROCEDURE ProveBody(body : Ptr to a BodyRec) : BOOLEAN;
       VAR a : Ptr to an AtomRec;
           t : BOOLEAN;
       BEGIN
          t := TRUE;
          a := first atom in body;
          WHILE (a <> NIL) AND (t) DO
             t := ProveClause (HeadRec for atom a);
             a := next atom in body;
          END;
          RETURN t;
       END ProveBody;

    BEGIN
       t := FALSE;
       b := first BodyRec for the goal symbol;
       WHILE (b <> NIL) AND ( NOT t) DO
          t := ProveBody(b);
```

```
            b := next BodyRec for this symbol;
         END;
         RETURN t;
      END ProveClause;

BEGIN
   IF(goal=NIL)OR(NOT ProveClause(goal)) THEN WrStr('NO')
                                         ELSE WrStr('YES'); END;
END Prove;
```

2 FIRST-ORDER LOGIC

2.1 Objectives

This chapter extends the refutation methods developed in the previous chapter to 1st order logic.

It is shown that a set S of 1st-order clauses is unsatisfiable if and only if it is false under all interpretations over the Herbrand universe (called Herbrand interpretations). Moreover, S is unsatisfiable if and only if a FINITE set of ground (ie variable-free) instances of clauses in S is unsatisfiable, if and only if S has a (finite) failure tree. As before, resolution refutation attempts to collapse the failure tree of S into a one-node tree. The resolution procedure given for propositional clauses can be LIFTED to non-ground clauses by unification - if C_1 and C_2 are two clauses whose instances C_1' and C_2' have resolvent C_3', then C_1 and C_2 have resolvent C_3 such that C_3' is an instance of C_3. This result is used to demonstrate the completeness of resolution refutations in the 1st order case using a proof similar to the one used for the propositional case.

2.2 Syntax

The alphabet of a 1st order language is

a. The set of logical connectives $LCON = \{\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg\}$
b. the set of quantifiers $QUANT = \{\exists, \forall\}$
c. the set of auxiliary symbols $AUX = \{(,)\}$
d. a countably infinite set of variables $VAR = \{v_0, v_1, ...\}$
e. a set L of non-logical symbols consisting of
   i. a countable, possibly empty, set of **FUNCTION SYMBOLS**, (or **FUNCTORS**) $FS = \{f_0, f_1, ...\}$ together with a rank function $\Re: FS \rightarrow \mathbb{Z}^+$. The number $\Re(f_i)$ is called the **ARITY** of f_i.
   ii. a countable, non-empty, set of **PREDICATE SYMBOLS** $PS = \{P_0, P_1, ...\}$, together with a rank function $\Re: PS \rightarrow \mathbb{Z}^+$. $\Re(P_i)$ is called the **ARITY** of P_i. Predicate symbols of arity 0 are propositional symbols.

The sets *FS* and *PS* are disjoint.

For convenience we sometimes refer to the set of constant symbols $CS \subseteq FS$, the set of function symbols of arity 0.

In the sequel we let the symbols

u,v,w (possibly subscripted) range over the set VAR
f,g,h (possibly subscripted) range over the set FS
a,b,c (possibly subscripted) range over the set CS
and P,Q,R (possibly subscripted) range over the set PS

TERMS. The set *TERM* of expressions of sort **TERM** is informally defined by:

a. a variable symbol $v \in VAR$ is a term,
b. if $f \in FS$ is a function symbol and $\Re(f)=n$, then the expression $f(t_1,...,t_n)$, where t_1 to t_n are n terms, is a term. If $\Re(f)=0$, ie $f \in CS$, we write f instead of f().

ATOMIC FORMULAE. The set *AFORM* of expressions of sort **ATOMIC FORMULA** is defined by

a. if $P \in PS$ is a predicate symbol and $\Re(P)=n$, then the expression $P(t_1,...,t_n)$, where t_1 to t_n are n terms, is an atomic formula. If $\Re(P)=0$, we write P instead of P().

LITERALS. The set *LIT* of expressions of sort **LITERAL** consists of all atomic formulae A and their negation, $\neg A$.

WELL-FORMED (1^{ST} ORDER) FORMULAE. The set *WFF* of well-formed (1^{ST} order) formulae is defined by

a. all atomic formulae are well-formed formulae,
b. if A is a well-formed formula, then so is $\neg A$,
c. if A,B are well-formed formulae, then so are $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$ and $(A \Leftrightarrow B)$,
d. for any $v_i \in VAR$ and $A \in WFF$, $\forall v_i A$ and $\exists v_i A$ are well-formed formulae.

2.3 Standard Form

2.3:a DEFN: *(PRENEX NORMAL FORM - PNF).*
A formula F is said to be in PNF iff F has the form

$$\mathbf{Q_1 v_1 ... Q_n v_n\ M[v_1,...,v_n]}$$

where $Q_i \in$ QUANT, $v_i \in$ VAR, and M is a quantifier-free formula with variables v_1 to v_n. $Q_1 v_1 ... Q_n v_n$ is called the **PREFIX** of F, M is called the **MATRIX** of F.

2.3:b THEOREM:
For every formula F there exists a formula F^* in PNF such that $F \equiv F^*$ (ie logically equivalent).

For a constructive proof see for example [CHA75 pp37ff][GAL87 pp307ff].

2.3:c THEOREM: *(SKOLEM)*
Let F=PM be a formula in PNF, where P is a prefix of quantified variables and M is a matrix. Then there exists a formula S_F=P'M', where P' is a prefix of universally quantified variables and M' is a matrix, such that F is satisfiable iff S_F is satisfiable. We say S_F is the **Skolem standard form** of F.

Proof and construction given in [CHA75 pp46ff][GAL87 pp357ff]

NOTE that F and S_F are not in general equivalent.

EXAMPLE:

Let $F = \forall x \exists y\ (P(x) \wedge Q(y))$
Let $G = \forall x\ (P(x) \wedge Q(f(x))$

Suppose G is satisfiable. Then, for all x we can find a y=f(x) such that Q(f(x)) is true, and therefore Q(y) is true. Hence F is also satisfiable. If G is inconsistent then there is an x for which no y=f(x) exists which makes Q(f(x)) true. Hence F is also inconsistent.

Conversely, if F is satisfiable, then for all x there exists a y such that Q(y) is true. But then we can let f(x)=y, thus making Q(f(x)) also true. Hence G is satisfiable. On the other hand, if F is inconsistent, then for some x there is no y which makes Q(y) true. Therefore, no matter what value is assigned to f(x), Q(f(x)) cannot be true. Hence F is also inconsistent.

Let W be a wff and F=PM an equivalent formula in PNF. Let S_F=P'M' be its Skolem standard form. Since M' is a quantifier-free formula, we can transform M into clausal (conjunctive normal) form and write S_F=P'C (see section 1.5:c). Moreover, since all variables in C are universally quantified, we can omit the prefix altogether and merely write C. Then the clause C has the property that W is satisfiable if and only if C is satisfiable.

Henceforth assume that all formulae are of the form

$$\mathbf{F ::= L_1 \vee L_2 \vee ... \vee L_n}$$

where L_i are literals and all variables are universally quantified. In the sequel it will be assumed that clauses have disjoint variable sets. This requirement is easily met by renaming variables. As for the propositional calculus, we will need to restrict our attention to (universally quantified) predicate Horn clauses when considering SLD resolution later on.

2.4 Semantics

2.4:a DEFN: *(INTERPRETATION)*
An interpretation I is a triple $<D,I_v,I_c>$, where

a. D, the **DOMAIN OF INTERPRETATION** (or simply **DOMAIN**), is a non-empty set.

b. I_v is a function from the set *VAR* of variables to the set D.

c. I_c is a function which
   i. maps each predicate symbol $P \in PS$ of arity n to a function $I_c(P):D^n \rightarrow BOOLEAN$, and
   ii. maps each function symbol $f \in FS$ of arity n to a function $I_c(f):D^n \rightarrow D$.

Let S be a 1st-order formula, and D the domain of interpretation. The **BASE OF S WITH RESPECT TO D**, B_{SD}, is the set $\{P(t_1,...,t_n) : P \in PS, t_i \in D\}$.

2.5 The Herbrand Universe And Base

2.5:a DEFN: *(GROUND TERM)*.
A term containing no variables is called a ground term.

2.5:b DEFN: *(GROUND LITERAL)*.
A literal all of whose arguments are ground terms is called a ground literal.

2.5:c DEFN: *(GROUND CLAUSE)*.
A clause $C = \{l_1,...,l_n\}$ is said to be a ground clause if $\forall l_i \in C$, l_i is a ground literal.

Let S be a finite set of clauses and H_0 the set of constant symbols (0-ary function symbols) appearing in S (or $\{\tau\}$ if S has no constant symbols). Let H_{i+1} ($i \geq 0$) be defined recursively by

$$\mathbf{H_{i+1} = H_i \cup \{ f(t_1,...,t_n) : t_1,...,t_n \in H_i, \Re(f)=n\}}$$

2.5:d DEFN: *(HERBRAND UNIVERSE)*.
The set $\lim(i\rightarrow\infty) H_i$ of a (finite) set of clauses S is called the **HERBRAND UNIVERSE** (or Domain) of S. The Herbrand Universe of a set S is written H_S, or simply H.

EXAMPLE:

Let S = { {P(a)} , {P(u), ¬P(g(v))} }

Then $H_0 = \{a\}$
$H_1 = \{a, g(a)\}$
$H_2 = \{a, g(a), g(g(a))\}$
$H_S = \{a, g(a), g(g(a)), g(g(g(a))), ... \}$

Clearly, H_S is finite iff S contains only function symbols of arity 0 (ie constants). In that case, H_S is merely the set of constants in S, which is finite since we are primarily interested in finite sets of clauses.

2.5:e DEFN: *(HERBRAND BASE).*

Let S be a (finite) set of clauses. The set of ground atoms $B_S = \{P(t_1,...,t_n)$: for all predicates P in S and t_i in $H_S\}$ is called the Herbrand Base (or atom set) of S.

EXAMPLE:

Let S = { {P(u)} , {Q(a,f(v)} }

Then $H_S = \{a, f(a), f(f(a)), f(f(f(a))), ... \}$
and $B_S = \{P(a), Q(a,a), P(f(a)), Q(a,f(a)), P(f(f(a))), ... \}$

Note that if S is finite, then B_s is enumerable

2.5:f DEFN: *(GROUND INSTANCE).*

Let S be a (finite) set of clauses. A ground instance of a clause $C \in S$ is obtained by replacing variables in C by members of H_S.

2.6 Herbrand Interpretations And Semantic Trees

2.6:a DEFN: *(HERBRAND-INTERPRETATIONS).*

An interpretation I_H of S is said to be a Herbrand Interpretation (H-Interpretation) if

a. the interpretation domain is the Herbrand Universe H
b. I_H maps all constants in S onto themselves
c. for every function symbol f of arity n in S, f is assigned a function $f':H^n \rightarrow H$ which maps the n-tuple $(t_1,...,t_n) \in H^n$ to $f(t_1,...,t_n) \in H$.

Let $B_S = \{A_1,A_2,...\}$ be the Herbrand base of a set of clauses S. A Herbrand interpretation I_H can be conveniently represented by a set $\{m_1,m_2,...\}$, such that $m_i=A_i$ if $I_H(A_i)$=TRUE, and $m_i=\neg A_i$ if $I_H(A_i)$=FALSE. Each such interpretation is a path in a semantic tree T_s of depth $|B_s|$.

EXAMPLE:

Let S = { {P(u)} , {Q(a,f(v)} }

Then $H_S = \{a, f(a), f(f(a)), f(f(f(a))), ... \}$
and $B_S = \{P(a), Q(a,a), P(f(a)), Q(a,f(a)), P(f(f(a))), ... \}$

The semantic tree T_s for S is then the infinite tree

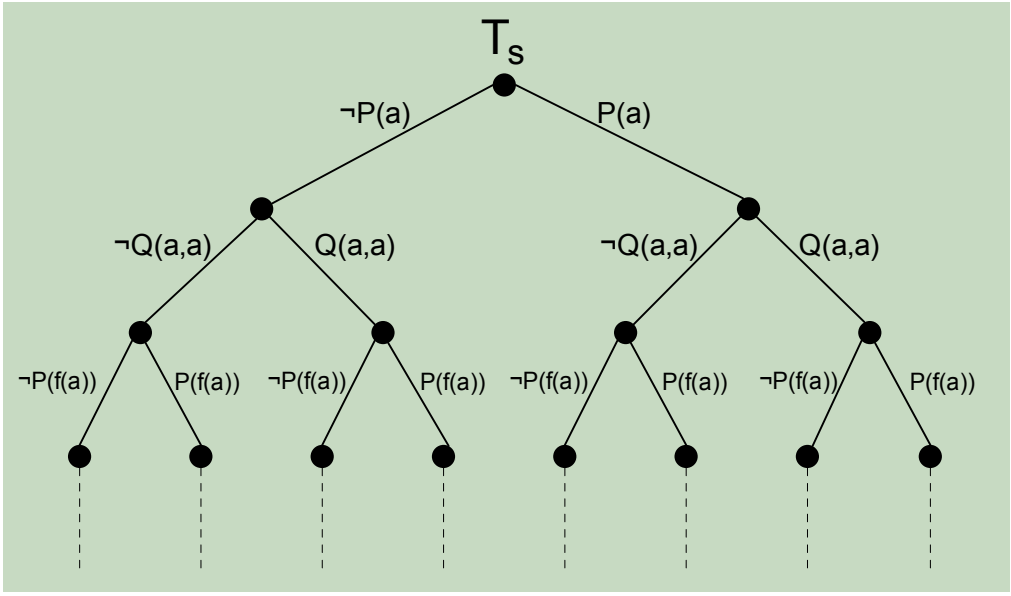


Figure 12: Semantic Tree showing a partial interpretation

The path shown in the diagram corresponds to the interpretation $\{P(a),Q(a,a),\neg P(f(a)),...\}$. For some node N in T_s, a path from the root to N corresponds to a partial interpretation I_N. As with

propositional semantic trees, we can define a FAILURE NODE in a semantic tree to be a node N such that the (partial) interpretation I_N falsifies some ground instance of a clause of S but I_M satisfies S for every ancestor node M of N.

2.6:b DEFN:

Given an interpretation I over a domain D, an H-interpretation (ie an interpretation over the Herbrand Universe H) I^* corresponding to I can be constructed as follows:

a. define a function $\mathscr{H}$:H→D, mapping each element h_i in the Herbrand universe to an element $\mathscr{H}(h_i)$ in the domain D.

b. define I^* such that $I^*(P(h_1,...,h_n))$=TRUE iff $I(P(\mathscr{H}(h_1),...,\mathscr{H}(h_n))$=TRUE for $h_1...h_n$ elements of H.

NOTE that if S contained no constant symbols, then the element τ in H_0 must be mapped onto every element of D generating |D| H-interpretations corresponding to I.

Clearly, if an interpretation I over a domain D satisfies a set of clauses S, then any H-interpretation I^* corresponding to I also satisfies S, for if $P(d_1,...,d_n)$ is true under I, then $P(h_1,...,h_n)$, where $d_i=\mathscr{H}(h_i)$, is by definition true under I^*.

2.6:c THEOREM:

A set of clauses S is unsatisfiable if and only if it is false under all H-interpretations.

PROOF:

By definition, if S is unsatisfiable then it is false under all interpretations, including H-interpretations. Conversely, assume S is false under all H-interpretations but is nevertheless satisfiable. Then there exists an interpretation I over a domain D≠H which satisfies S. But then the H-interpretation I^* corresponding to I must satisfy S, contradicting our assumption that S is false under all H-interpretations. Hence, S must be unsatisfiable.

2.7 Herbrand's Theorem

2.7:a THEOREM: *(HERBRAND)*.

A (finite) set of clauses S is unsatisfiable if and only if some **FINITE** set of ground instances of clauses in S is unsatisfiable.

PROOF:

(⇒) Suppose S is unsatisfiable. Then every Herbrand interpretation falsifies S, and so every path in the semantic tree must terminate in a failure node and is therefore of finite length. Hence there exists a finite failure tree FT_s of S. Since the number of failure nodes in FT_s is finite, and the number of ground literals in FT_s is finite, the set S' of ground instances of S falsified at each failure node must also be finite.

(⇐) Conversely, let S' be a finite unsatisfiable set of ground instances of clauses of S. For each interpretation I of S, let I' be the restriction of I to S'. Since every interpretation I' falsifies some clause in S', then every interpretation I must also falsify some clause in S'. Hence every path in the semantic tree of S T_s must terminate in a failure node, and therefore S has a failure tree. Then by lemma 1.8:d S must be unsatisfiable.

Herbrand's theorem has formed the basis of many automatic refutation procedures, in particular 'level-saturation' procedures in work by Gilmore (1960), and Davis and Putnam (1960) (see [CHA75 pp62ff],[ROB65 p27] and [LOV84 pp7ff]).

2.8 Substitutions

2.8:a DEFN: *(SUBSTITUTION)*.

A substitution is a homomorphism θ:*TERM* → *TERM*, defined by:

a. $\theta(c) = c, \forall c \in CS$

b. $\theta(v) = t$, $\forall v \in VAR$, and some $t \in TERM$
c. if $f \in FS$ and $\Re(f)=n$, then $\theta(f(t_1,...,t_n)) = f(\theta(t_1),..., \theta(t_n))$.

2.8:b DEFN: *(SUPPORT OF A SUBSTITUTION)*.
The set $V(\theta) = \{v \in VAR : \theta(v) \neq v\}$ is called the support of the substitution θ (or the set of **ACTIVE VARIABLES** of θ).

We will denote a substitution θ with support set $\{v_1,v_2,...\}$ such that $\theta(v_i)=t_i$ for some term $t_i \neq v_i$ by $\{t_1/v_1, t_2/v_2,...\}$. We will only be interested in finite substitutions, ie substitutions with finite support sets $\{v_1,...,v_k\}$.

2.8:c DEFN: *(INSTANTIATION)*.
Let $F \in$ *WFF* be a formula and $\theta=\{t_1/v_1,..., t_k/v_k\}$ be a substitution. Then the **INSTANTIATION** OF F BY θ, written $F\theta$, is the operation of applying θ to all terms in F. $F\theta$ is said to be an **INSTANCE** of F.

2.8:d DEFN: *(GROUND SUBSTITUTION)*.
A substitution $\theta=\{t_1/v_1,...,t_k/v_k\}$ is said to be a ground substitution if all t_i are ground terms. $F\theta$ is then said to be a **GROUND INSTANCE** of formula F.

2.8:e DEFN: *(COMPOSITION OF SUBSTITUTIONS)*.
The composition $\theta\lambda$ of two substitutions

$\theta=\{t_1/v_1,...,t_k/v_k\}$ and $\lambda=\{s_1/u_1,...,s_n/u_n\}$

is $\theta' \cup \lambda'$, where

$\theta' = \{t_i\lambda/v_i \ : \ t_i\lambda \neq v_i\}$
$\lambda' = \{s_j/u_j \ : \ u_j \neq v_i \text{ for all } v_i \text{ in } \theta\}$

EXAMPLE:

Let $\theta = \{f(x)/y, z/x, a/w\}$
$\lambda = \{a/x, b/y, w/z\}$

Then $f(x)\lambda/y = f(a)/y$
$z\lambda/x = w/z$
$a\lambda/w = a/w$

and so

$\theta' = \{f(a)/y, w/z, a/w\}$
$\lambda' = \{w/z\}$
$\theta\lambda = \theta' \cup \lambda' = \{f(a)/y, w/z, a/w\}$

Note

a. composition of substitutions is associative, ie $(\theta\lambda)\mu = \theta(\lambda\mu)$
b. the empty substitution ε is both right and left identity, so that $\varepsilon\theta = \theta\varepsilon = \theta$

2.9 Unification

2.9:a DEFN: *(UNIFIER)*.
A substitution θ is said to **UNIFY** a finite, non-empty set of well-formed formulae $S=\{S_1,...,S_n\}$ if $S_1\theta = S_2\theta =...=S_n\theta$. θ is said to be a **UNIFIER** of the set S, and S is stb **UNIFIABLE** if there exists a unifier for it. $S_i\theta$ is said to be a **COMMON INSTANCE** of the set S for any $S_i \in S$.

2.9:b DEFN: *(MOST GENERAL UNIFIER)*.
A unifier μ for a set $S=\{S_1,...,S_n\}$ is a **MOST GENERAL UNIFIER** (MGU) if, for each unifier θ of S there exists a substitution λ (possibly ε) such that $\theta=\mu\lambda$.

MGUs are unique when they exist in that, if μ and λ are two MGUs for a set S, then there exists a bijective substitution σ such that $\mu=\lambda\sigma$ and $\sigma(v) \in VAR$ for all v in the support set of σ. In other words, σ merely renames variables. (for a proof see GALLIER p.383 ff).

2.9:c THEOREM: (UNIFICATION THEOREM).

If a finite, non-empty set S of WFFs is unifiable, then S has a MGU μ_s. Moreover, μ_s is unique down to renaming of variables and, for all unifiers θ_s of S, there exists a substitution σ such that $\theta_s=\mu_s\sigma$.

For a proof see [ROB65 pp33-34].

2.10 Representation Of Terms

A term may be understood to be a linear representation of an oriented tree (more precisely a directed acyclic graph, because nodes may share common subgraphs, but the tree representation is more common and easier to handle).

The tree corresponding to a term t is defined recursively as follows:

a. the tree representation of a variable is a single node
b. if t is a function symbol f with arity n, then t is represented as a tree with root f and n subtrees, one for each argument term of f.

EXAMPLE:

The term t = f(g(a,x), y, h(x,y,g(x))) has tree representation:

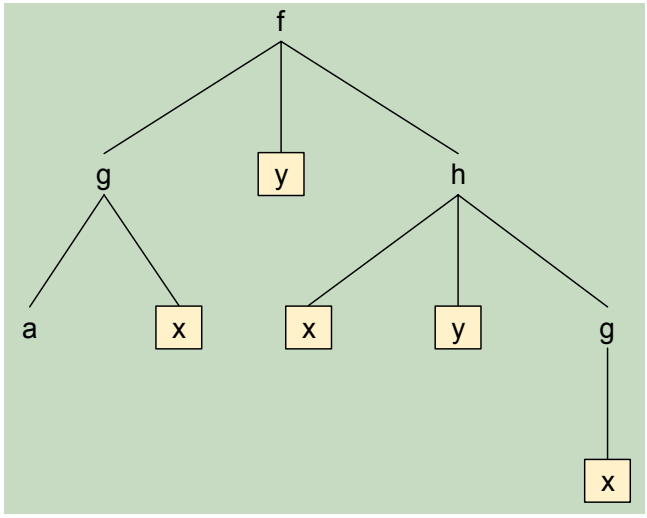


Figure 13: Term Tree

The boxed leaves denote variables.

2.11 Unification Algorithm

In unifying two terms T_1, T_2, we traverse the two term trees in parallel in pre-order, noting disagreement at each node. A disagreement between nodes N_1,N_2 of T_1,T_2 respectively is **REPARABLE** if at least one of N_1,N_2 (say N_1) is a free variable. The substitution $\{N_2/N_1\}$ will repair the disagreement. The substitution must be applied to the two term trees before traversal continues, since all occurrences of N_1 must be instantiated to N_2. The MGU of T_1 and T_2 is the composition of all the substitutions required to repair node-disagreements. If a disagreement is not reparable (as when N_1 and N_2 are different functors), then the terms are not unifiable.

It is possible for a free variable to be instantiated to a term containing the variable as a subterm, creating infinite subtrees (or cyclic graphs in the directed graph representation of terms). For example consider the two terms

$\mathbf{T_1 = f(x,x)}$
$\mathbf{T_2 = f(y,g(y))}$

Since x and y are both free variables, the substitution {x/y} is generated. For the second subterm, x is still free (since it is bound to a free variable), and so the substitution {g(y)/x} is generated. This gives the MGU {x/y}{g(y)/x} = {g(y)/y}, which results in the common term instance f(g(g(g(...., g(g(g(....), which is an infinite tree (or a cyclic graph).

To avoid this, the unification algorithm should check that in a substitution {t/v}, v is not a node of t. This check, called the **OCCURS CHECK,** is very expensive, and is often omitted from implementations of unification algorithms (eg in Prolog) on the grounds that such cyclic structures occur very rarely in practice, and so do not warrant the increased complexity associated with the occurs check.

The following algorithm computes the MGU of two terms, returning FALSE if the terms are not unifiable. The procedure is called with two terms and the empty substitution ε as arguments, and returns the MGU of the terms if they are unifiable.

```
PROCEDURE Unify (T1,T2 : TERMS; VAR mgu : substitution) : BOOLEAN;
BEGIN
   Let N1 := root node of T1
   Let N2 := root node of T2
   IF N1≠N2 THEN
      IF IsVar(N1) OR IsVar(N2) THEN
         IF IsVar(N1) THEN RETURN Instantiate(N1,T2,mgu)
         ELSE              RETURN Instantiate(N2,T1,mgu)
      ELSE RETURN FALSE;
   ELSE
   (* N1 and N2 are same functor. Unify their arguments. *)
   FOR each subtree S1,S2 of T1,T2 respectively DO
      IF NOT Unify(S1,S2,mgu) THEN RETURN FALSE
END Unify;
```

Procedure *Instantiate* attempts to instantiate the variable (v) passed as a first argument to the term tree (t) passed as the second argument. If successful (ie if $v \notin t$), the new substitution {t/v} is composed with the partial MGU constructed so far, and the two term trees are updated to remove the disagreement.

2.12 Resolution In 1st Order Logic

The only difference between a resolution refutation in propositional logic and in 1st order logic is in the way resolvents are formed. Resolution on 1st order clauses is defined in terms of unification. Let C_1 and C_2 be two clauses (with disjoint variable sets).

$\mathbf{C_1 = \{L_1,..,L_m\}}$
$\mathbf{C_2 = \{M_1,..,M_n\}}$

Let L_i=P and M_j=¬Q such that P and Q have a most general unifier μ. Then the **BINARY RESOLVENT** of C_1 and C_2 on L_i and M_j is the clause

$\mathbf{C_R = (C_1 \backslash \{L_i\})\mu \cup (C_2 \backslash \{M_j\})\mu}$

EXAMPLE

Let C_1 = a(Y) :- b(t(Y)).
C_2 = c(X) :- a(d(X)).

where X and Y are variables.

Then the MGU of a(Y) and a(d(X)) is

μ = {d(X)/Y},

and the resolvent of C_1 and C_2 is

$$\neg b(t(Y))\mu \cup c(X)\mu, = c(X) :- b(t(d(X))).$$

2.13 Completeness of the Resolution Principle (1st Order Case)

The proof of the completeness of the resolution principle in 1st order case will closely follow that for propositional logic (see section 1.11). As before, we show that if a set of clauses S has a failure tree FT_s, then resolution collapses FT_s to a single-node tree □.

2.13:a LEMMA: *(LIFTING LEMMA).*

Let C_1 and C_2 be two predicate clauses (with disjoint variable sets) whose instances C_1' and C_2' have resolvent C_3'. Then C_1 and C_2 have resolvent C_3 such that C_3' is an instance of C_3.

For a proof see [CHA73 pp84-85][GAL87 pp400-403].

2.13:b THEOREM: *(COMPLETENESS OF RESOLUTION - 1st ORDER CASE).*

A set S of 1st-order clauses is unsatisfiable if and only if the empty clause □ can be deduced from S by resolution.

PROOF:

The proof follows that of theorem 1.11:a, except that the partial interpretations I_j and I_k at the failure nodes i and k now falsify two ground instances C_j' and C_k' of the clauses C_j and C_k of S. Hence the resolvent C_R' of C_j' and C_k' is falsified at or before the inference node i (the parent of nodes j and k). But by the lifting lemma, C_R' is an instance of C_R, the resolvent of C_j and C_k. Hence, the set $S \cup \{C_R\}$ has a smaller failure tree than S, since an instance of C_R is falsified at or above node i.

2.14 Logic Programming

A 1st-order logic-program is a set S of definite 1st-order Horn-clauses representing **FACTS** and **RULES,** together with a negative Horn clause G called the **GOAL.** The goal represents a negated **QUERY** Q, which has the form

$$\mathbf{Q = \exists v_1,..,v_n(L_1 \wedge .. \wedge L_k).}$$

where $v_1,..,v_n$ are called the **OUTPUT VARIABLES.**

Thus the goal is a formula

$$\mathbf{\forall v_1,..,v_n(\neg L_1 \vee .. \vee \neg L_k)}$$

which is a universally quantified Horn-clause

$$\mathbf{:- L_n,..,L_k.}$$

It is required to find terms $t_1,..,t_n$ such that

$$\mathbf{S \Rightarrow Q\{t_1/v_1,..,t_n/v_n\}}$$

The substitution $\{t_1/v_1,..,t_n/v_n\}$ is called the **ANSWER *SUBSTITUTION***, and a logic interpreter usually attempts to find all such answer substitutions.

The requirement that clauses have disjoint sets of variables is usually enforced by limiting the scope of all variables to the clause in which they occur.

The refutation procedure for 1st-order Horn-logic programs is very similar to that for propositional programs. Using SLD-resolution with a depth-first search strategy, the refutation procedure is the following procedure (Satisfy) called with G and the empty substitution ε as

parameters. If Satisfy terminates successfully, the output substitution Θ contains the answer substitution as a subset.

```
PROCEDURE Satisfy (G : goal; VAR Θ : substitution) : BOOLEAN;
VAR σ,μ : substitution;
BEGIN
   IF G=□ THEN RETURN TRUE;
   WHILE (C := SELECT(FIRST(G),μ)) ≠ NIL
      σ := θμ;
      IF Satisfy((BODY(C)+REST(G))μ,σ) THEN
         θ := σ;
         RETURN TRUE;
      END;
   END;
   RETURN FALSE;
END Satisfy;
```

The **SELECT** procedure must now perform unification in order to determine the set of candidate clauses for a literal. The MGU μ returned by **SELECT** is used to instantiate the two parent clauses C and G, producing C' and G'. The output substitution Θ is the composition of all substitutions made in the course of the refutation.

3 PROLOG

3.1 Objectives

This chapter outlines the implementation of Prolog. Following a brief review of the Prolog language, the interpretation strategy is discussed. The representation of terms constructed during unification in structure-sharing and non-structure-sharing systems is compared, and a basic optimization technique which exploits determinism in a Prolog program (deterministic-frame optimization, DFO) is outlined. Two other optimization techniques based on DFO, last-call optimization and tail-recursion optimization, are briefly described.

3.2 The Language - Syntax And Terminology

The following is an overview of the syntax of the basic Prolog language. For a description of the full language see [CLO81].

Program

A Prolog program is a sequence of clauses.

Clause

A clause is made up of a **HEAD** and a **BODY.** The body consists of a (possibly empty) sequence of **GOALS.** A clause is written in the form

a :- b,c,d.

where **a** is the head, and **b,c,d** are the goals (or **PROCEDURE CALLS**) making up the body of the clause. The head and goals of a clause are examples of **TERMS.**

If the body is the empty sequence, then the clause is called is called an **ASSERTION** (or **UNIT CLAUSE**) and is simply written as

a.

A clause which does not have a head is called a **GOAL STATEMENT**, and is written

:- b,c,d.

Terms

Terms may be **SIMPLE** or **COMPOUND**.

a. SIMPLE TERMS

Simple terms are either **VARIABLES** or **CONSTANTS**.

1. variables

   A variable is an identifier beginning with an uppercase letter or the underscore character, _ . The identifier consisting solely of the underscore character is called the ANONYMOUS VARIABLE.

2. constants

   A constant is either an ATOM or an INTEGER. An integer is any sequence of characters from the set {'0'..'9'}, optionally preceded by one of '+' or '-'. An atom is any sequence of characters not confusable with either a variable or an integer. The following are all valid atoms:

   name, 'Name', 'A NAME', ==

b. COMPOUND TERMS

A compound terms is a structure consisting of a **FUNCTOR** together with a list of one or more terms called **ARGUMENTS**.

A functor is an ordered pair **NAME/ARITY**, where name is an atom, and arity (or **RANK**) is a positive integer denoting the number of arguments associated with the functor. A compound term is written as:

f(a,b,c)

where f/3 is the functor (or **PRINCIPLE FUNCTORS**), and a,b,c are its arguments. Note that a constant is considered to be a functor of arity 0.

Literals

Terms which appear as the head or goals of a clause are called **LITERALS** (sometimes **BOOLEAN TERMS**). In general, literals are not allowed to be variable or integer terms.

Predicates

The functor of a literal is called a **PREDICATE**.

3.3 Semantics

Declarative Semantics

The declarative semantics of Prolog are the semantics of Horn-clause programs under SLD-resolution, with the database of definite clauses modelling facts and rules, and a single negative clause (the goal) representing a negated query.

Since SLD-resolution is an incomplete procedure (in that it does not specify a search strategy, as explained in the Chapter 2), the declarative semantics of Prolog do not fully describe the language.

Procedural Semantics

The procedural semantics of Prolog are the semantics of Horn-clause programs under SLD-resolution with a depth-first search strategy. In this sense, the procedural semantics of Prolog are complete since they comprise the semantics of the search strategy. Because the search strategy presupposes a sequential Von-Neumann architecture, problems have been encountered in preserving the procedural semantics in parallel-implementations of Prolog (see for example [TIC89]).

In the procedural model of Prolog (first proposed by R.A.Kowalski [EMD76][NIL84]) the set of clauses whose head have the same predicate are viewed as a non-deterministic **PROCEDURE** to be **INVOKED** (or **ENTERED**) when that predicate is encountered in a goal literal. The goal literals in a clause body are in this sense **PROCEDURE CALLS**, resolution is viewed as a procedure invocation, and unification as a parameter transfer mechanism.

The clauses in a procedure are considered to be ordered from top to bottom, while goals in a clause body are ordered from left to right. This ordering constitutes control information which is superimposed on the logic component of Prolog, and which defines the depth-first search strategy imposed on the SLD-resolution mechanism.

3.4 Control Mechanism

Prolog uses SLD-resolution with a depth-first search, and so the execution mechanism is very similar to the *Satisfy* procedure given in section 2.14, with the input goal statement as the first centre clause, and the leftmost literal in the goal as the first selected literal. The interpreting algorithm for Prolog programs was first fully described in [EMD84].

The main states of the interpreter are

1. **INITIALIZE** - make the input goal the current procedure.

2. **PROCEDURE ENTRY** - make the body of the selected procedure the current goal list, and prepare to start executing the first call in this list.
3. **SELECT CALL** - select the call to try next. This is the first literal in the current goal list. If the current goal list is empty (ie the current procedure has no body) then exit the procedure (goto step 5).
4. **SELECT PROCEDURE** - select, from among the candidate clauses for the selected call, the first clause whose head unifies with the selected call (we call such a clause the **RESPONDING PROCEDURE**), and goto step 2. If no procedure responds, then the interpreter must backtrack (goto step 6).
5. **PROCEDURE EXIT** - If the current procedure has a parent with some calls pending, then make the parent goal the current goal, with the next call in the parent clause the selected call, and goto step 4.

   Otherwise the original (input) goal has been solved. Output solution and backtrack (goto step 6).
6. **BACKTRACKING** - find the most recent call for which some candidate clauses remain untried. If there is no such call, then execution terminates. Otherwise, make this the current call and the set of untried candidates the new candidate set, and goto step 4.

3.5 Activation Frames

When a procedure is entered, an ***ACTIVATION FRAME*** (**FRAME** for short) is created and pushed onto the runtime stack. Frames are similar to procedure activation records in procedural languages, except that frames may not be popped on exit from a procedure because of the possibility of backtracking.

We can distinguish between **DETERMINISTIC** and **NON-DETERMINISTIC** frames [BRU84a p260]. A deterministic frame is a frame corresponding to the activation of a procedure for which no untried candidate clauses remain. A deterministic frame represents a call which cannot be reactivated, since all alternative solutions have been exhausted. A non-deterministic frame corresponds to the activation of a procedure for which there remain some untried candidates. Non-deterministic calls may be reactivated by backtracking, and for this reason are sometimes are called **BACKTRACKPOINTS**. The interpreter maintains non-deterministic frames on a linked list, which is implemented by threading the non-deterministic frames on the frame stack and maintaining a pointer to the most recent backtrackpoint.

Each frame is made up of two sections, one for control information (the **CONTROL VECTOR**), and one for variable bindings (the **ENVIRONMENT VECTOR**).

Control Vector

Call: pointer to the selected call which invoked the procedure.

Parent: pointer to the frame recording entry into the procedure containing the call which activated this procedure.

NextCand: pointer to the next untried candidate clause for the current call, (or NIL for a deterministic call).

BTPoint: Backtrack point in effect at time frame was created.

Environment Vector

Env: Array recording variable instantiations (or bindings) made during unification. Each cell of the array represents one variable in the activated procedure.

The interpreter also maintains a few state variables to keep track of the execution state. These include:

CrntCall: This is analogous to the instruction pointer in a hardware processor, and points to the selected (or current) call.

CrntParent: pointer to the frame for the procedure containing CrntCall as one of its subgoals.

CrntProc: Pointer to the procedure responding to CrntCall. This is the procedure which is to be invoked next.

CrntBTP: Current backtrack point. Pointer to the most recent non-deterministic frame. This corresponds to the most recent branch-node (or choice point) in the search tree. The contents of this variable are copied into the BTPoint field of the control vector of a frame.

3.6 The Interpretation Strategy

The introduction of frames and the four state variables makes the control mechanism more opaque. The control procedure outlined above can now be stated in finer detail. Following [KNU73 p.231] we use the notation **Field(Record pointer)** instead of the more cumbersome **RecordPtr^.Field**. Procedure *MakeFrame* creates a new procedure activation frame on the stack, returning a pointer to the new frame. The variable bindings in the new frame are initialized to the special value *FREE* to indicate that all variables are initially uninstantiated.

1. INITIALIZE

```
CrntCall := NIL
Parent     :=    NIL
BTPoint    :=    NIL
CrntProc   :=    goal
F := MakeFrame
```

2. PROCEDURE ENTRY

Initialize the control vector of the frame for this procedure activation. The variable bindings would already have been initialized by the unification algorithm. If there are untried candidate clauses for this procedure (ie this is a non-deterministic procedure call), then the new frame becomes the most recent backtrackpoint. The first call in the procedure becomes the new CrntCall.

```
Call(F)     := CrntCall
BTPoint(F)  := CrntBTP
Parent(F)   := Parent
NextCand(F) := next candidate clause following CrntProc
IF NextCand(F) <> NIL THEN
   CrntTP := F
END
Parent   := F
CrntCall := first call in CrntProc (NIL if CrntProc is an assertion)
```

3. SELECT CALL AND FIRST CANDIDATE CLAUSE

If the CrntCall is NIL, then the current procedure has no body, and may be exited immediately. Otherwise, set CrntProc to the first candidate clause for the current call.

```
IF CrntCall = NIL THEN
   goto step 5 (Procedure exit)
ELSE
   CrntProc := first clause for current call
END
```

4. SELECT RESPONDING PROCEDURE

Try each candidate clause in turn until one is found which unifies with (responds to) the current call. If no clause responds to the call, then the interpreter must backtrack. Otherwise make the responding clause the CrntProc and loop back to the procedure entry step. The

unification algorithm creates a new frame on the stack in which any variable bindings are recorded.

```
F := MakeFrame
WHILE (CrntProc <> NIL) AND NOT (Unify(CrntCall, CrntProc)) DO
       Undo any instantiations made during unification (this is
       called SHALLOW BACKTRACKING)
      CrntProc := next candidate clause
END
IF CrntProc = NIL THEN
   goto step 6 (Backtrack)
ELSE
   goto step 2
END
```

5. EXIT PROCEDURE

Exit current procedure and return to the parent procedure, which is to be resumed at its next (pending) call. If the current procedure has no parent, then this step represents an exit from the input goal clause, and so a solution is output and the interpreter backtracks in search of further solutions.

```
IF Parent <> NIL THEN
   Output solution and backtrack (goto step 6)
ELSE
   CrntCall := NextCall(Parent)
   Parent   := Parent(Parent)
   select responding clause for this call (goto step 3)
END
```

6. BACKTRACK

Backtrack to the most recent call which still has some untried candidate clauses (the CrntBTP). If there is no such call left (CrntBTP=NIL) then the interpreter halts. Otherwise all processing since the CrntBTP is abandoned and the most recent non-deterministic call reactivated with the next untried candidate clause.

```
IF CrntBTP = NIL THEN HALT
ELSE
   CrntProc := NextCand(CrntBTP)
   CrntCall := Call(CrntBTP)
   Parent   := Parent(CrntBTP)
   B        := BTPoint(CrntBTP)
   Pop all frames from CrntBTP onwards and undo all variable
    bindings made since.
   CrntBTP  := B
   goto step 4
END
```

3.7 Indexing Of Clauses

Clauses in the database are usually stored as tree structures and indexed using some indexing scheme. The purpose of indexing is to identify, at the time clauses are added to the database, those clauses which are in the candidate set of a call. Indexing attempts to preempt mismatches during unification (reducing shallow backtracking in step 4) by minimizing the number of candidate clauses which have to be tried. Reducing the number of alternative clauses which can respond to a call also reduces the non-determinacy of a Prolog program, since fewer choice-points in the search tree are created.

The simplest scheme (adopted in the original Marseille interpreter) is to index clauses on the head predicate. This reduces the candidate set for a call to those clauses which have the same functor in their head predicate as the functor in the call literal. Most Prologs also index clauses on the first argument of their head predicate, using the type (variable, functor, list) and arity of the first argument term of the predicate as a secondary key (see for example [WAR77]). More elaborate schemes based on static analysis of call patterns have also been suggested [DEB89a].

A related idea is to have the unification algorithm apply some dynamic (and relatively inexpensive) test on the clauses in the candidate set in order to filter out any candidates which could not possibly match the current call [BRU82 p.91]. When a call is successfully unified with the head of a candidate clause, the test is applied sequentially to the untried clauses in the candidate set. The first clause (if any) which passes the test becomes the next candidate clause, and is recorded in the *NextCand* field of the activation frame. The object of this scheme is to identify deterministic calls as early as possible, thereby enhancing the effectiveness of some optimization techniques (discussed below).

3.8 Implementing The ! Predicate

The ! predicate controls backtracking. Specifically, it makes all procedure calls since entry to the current procedure deterministic. Its implementation simply requires the interpreter to reset the current backtrack-point to the backtrack-point in effect when the current procedure was entered - which is recorded in the *BTPoint* field of the current procedure's *Parent* frame:

```
Cut: CrntBTP := BTPoint(Parent)
```

The effect of the ! predicate is to unlink one or more frames from the threaded list of non-deterministic frames on the frame stack.

3.9 The Binding Environment And The Trail

Variables in a clause are encoded as offsets into the *environment* vector within the activation frame for the clause. The environment records the variable instantiations made by unification in the course of generating a clause instance which responds to a call. All occurrences of a variable in a clause are mapped onto the same entry in the binding vector by the encoding scheme, so that they share the same binding. The binding environment within a frame will be represented as an array *env[0..numvars-1],* where *numvars* is the number of variables in the clause.

A clause instance is thus a pair **<skeleton,environment>**, where ***skeleton*** is the clause representation in the database, and the **environment** records the assignment for each variable in the clause.

Since variable bindings have to be undone on backtracking, a log of all variable instantiations made is kept on a stack structure called the ***TRAIL*** *(or* ***RESET LIST****)*. Only the address of variables needs to be recorded (since such variables must have been free prior to instantiation). Each entry on the trail is a pair **<frame,var>**, where frame is a pointer to a frame on the stack, and var is the index within the frame's environment vector of the variable instantiated.

When an activation frame is created, a pointer to the current top of trail is saved in the frame. Backtracking then simply has to reset all variables recorded on the trail following the trail pointer stored in the backtrackpoint frame.

A new variable *TrailTop* is required by the interpreter to keep track of the current top of the trail stack. A new field in the frame structure, *Trail,* will be used to save the position of *TrailTop* at the time of frame creation. The backtracking algorithm can now be stated more explicitly:

6. BACKTRACK

```
IF CrntBTP = NIL THEN HALT
ELSE
   CrntProc := NextCand(CrntBTP)
   CrntCall := Call(CrntBTP)
   Parent   := Parent(CrntBTP)
   B        := BTPoint(CrntBTP)
   T        := Trail(CrntBTP)
   FOR each entry <frame,var> on the trail from T to TrailTop DO
       env[var](frame) := free
   Pop all frames created since, and including, the frame
   pointed to by CrntBTP.

   CrntBTP  := B
```

```
    TrailTop := T
    goto step 4
END
```

Since backtracking pops all frames created since the last backtrackpoint, only the binding of variables in frames preceding the current backtrack frame need be recorded on the trail. Instantiations made to variables in more recent frames will be automatically undone when these frames are popped during the backtracking.

3.10 Structure-Sharing And Non Structure-Sharing Systems

Consider the unification of the two terms **a(f(g,h))** and **a(X)**, which (assuming X is free) requires the instantiation of X to **f(g,h)**. This instantiation can be easily effected by storing a pointer to the database code representing the term **f(g,h)** in the environment cell for variable X:

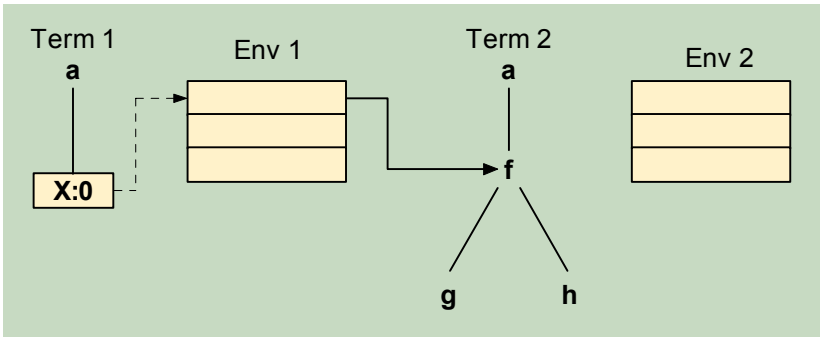


Figure 14: Unifying a(X) and a(f(g,h,))

But now consider the unification of the two terms

a(X)
and **a(f(G,H))**

Again assuming that X is free, the unification requires that X be instantiated to **f(G,H)**. However, simply storing a pointer in the environment cell for X to the representation of the term **f(G,H)** in the database will not work, since the instantiation must take into account the current (or future) bindings of the variables G and H. X should be instantiated NOT to the term **f(G,H)**, but to an **INSTANCE** of it. For example, if G is currently instantiated to g and H to h, then X should be bound to the term instance **f(g,h)**. Unification thus requires a mechanism for representing term instances, called **CONSTRUCTED TERMS**. Two common solutions to this problem exist, **STRUCTURE SHARING (SS)** and **NON-STRUCTURE SHARING (NSS)** (sometimes called **STRUCTURE COPYING**).

Structure Sharing

The idea behind structure sharing is that all instances of a term share a common prototype, differing only in the variable assignments. The term prototypes in the source program completely define the structure of any given instance of the term except for the value of the variables. Thus a term instance can be represented as a **<skeleton,environment>** pair, called a ***MOLECULE.*** The skeleton is a pointer to the database representation of the term (called the **SOURCE TERM**), which serves as the prototype for all instances of the term, while the environment is a pointer to a frame containing the bindings of the variables in the skeleton. Note that if the source term is a constant, there is no need to provide a binding environment, so that a constant is both a source term and a constructed term.

The term instance **a(f(g,h))** created during unification might be represented by a pointer to the skeleton **a(f(G,H))** and a pointer to an environment which binds G to g and H to h:

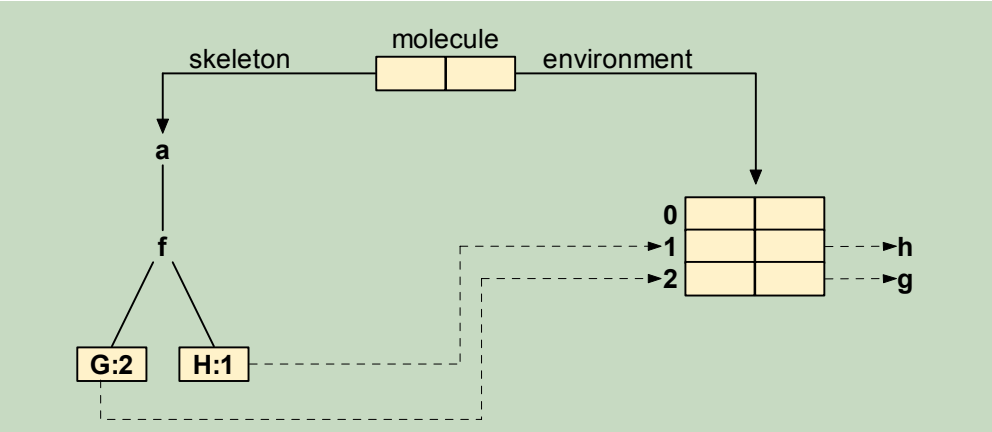


Figure 15: Representation of the term instance a(f(G,H)) in a structure-sharing system

When a (free) variable is instantiated to a term, a molecule representing the term instance is stored in the variable's cell in the environment vector. In structure sharing systems, the unification of the two terms **a(X)** and **a(f(G,H))** would generate the following binding for the variable X:

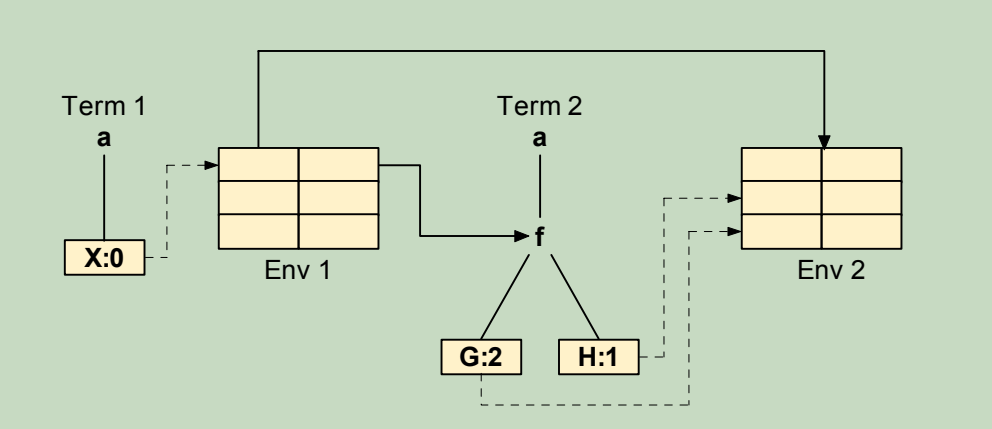


Figure 16: Unifying a(X) and a(f,(G,H)) in a structure-sharing system

Structure sharing economizes on space requirements for constructing complex terms since it adopts a *lazy* approach to term construction. On the other hand, accessing components of term instances may require considerable dereferencing.

Non-Structure Sharing

In *non-structure sharing* systems [MEL82], term instances are not represented as molecules. Instead, when a term is constructed during unification, a concrete (instantiated) copy of the term is created on the heap (called the **COPY STACK**, since it is organized as a stack structure). Variable bindings are not represented as **<skeleton,environment>** pairs, but as pointers to term instances. The scheme is very similar to that adopted for the creation of dynamic structures in procedural languages.

Figure 17 illustrates the unification of the two terms **a(Y,b)** and **a(f(X),X)** in a NSS system. Note how the variable Y is bound to the concrete instance **f(b)**, which is not a source term, but is constructed on the heap during unification.

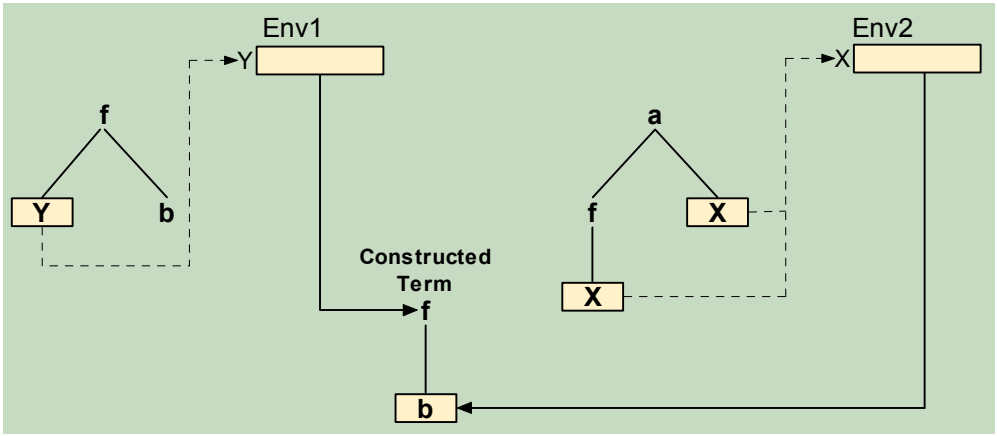


Figure 17: Unification of a(Y,b) and a(f(X),X)) in NSS systems

For comparison, Figure 18 shows the same unification in a SS system. Note how NSS systems avoid the need for **FORWARD BINDINGS** - binding a variable in one environment to a chronologically later environment. Bindings in NSS systems can always be oriented such that later frames reference earlier frames. This has important consequences for implementing optimization schemes discussed below.

On the other hand, SS systems cannot avoid forward bindings. Variable Y in the example requires *Env2* for the binding of variable X in the term skeleton **f(X)**. Consequently, *Env2* cannot be discarded without leaving dangling references in the earlier environment *Env1*. This creates problems when implementing optimization techniques aimed at conserving stack space (always a prime consideration in Prolog implementations) by discarding frames as early as possible.

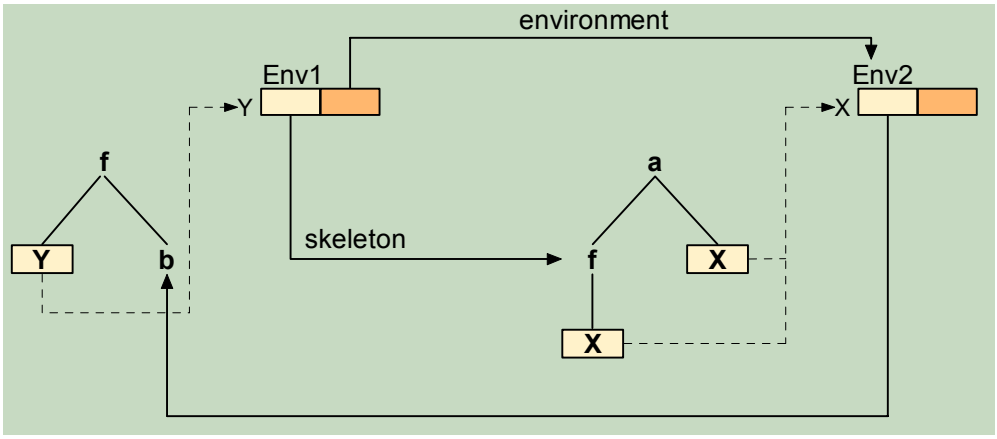


Figure 18: Unification of a(Y,b) and a(f(X),X) in a SS system

Although NSS generally incurs more space overhead than SS, accessing constructed terms is faster. Also, some optimizations techniques are easier to implement in a NSS system.

3.11 Deterministic-Frame Optimization

Most optimization techniques are concerned with conserving memory by discarding stack frames at the earliest opportunity. In procedural languages, procedure activation records are popped immediately a procedure exits, but in Prolog frames have to be retained for two reasons:

1. because of possible backtracking. A procedure's activation record contains information about the set of untried alternative candidates for the current call. If the frame is popped when the procedure exits, no backtracking is possible.
2. because of possible forward references to a frame. If variables in earlier frames are bound to the environment vector of the frame to be popped, then it is not possible to discard the frame without creating dangling pointers. This problem does not arise in procedural languages,

since variables kept in a procedure's activation record are purely local, and hence may be safely deallocated when the procedure terminates. In Prolog, however, variables in one procedure may be *exported* to a parent procedure as subterms. The problem is illustrated in Figure 19, where discarding the frame containing *Env2* will destroy the binding of Y in *Env1*.

Thus, a frame may be discarded on procedure exit if both the following conditions hold:

1. The frame is a deterministic frame (ie not a backtrackpoint). Deterministic frames are not targets for backtracking, and therefore do not contain any information which will be required to effect a backtrack operation.
2. No variables in earlier frames are forward bound to the environment vector of the frame to be discarded, either directly (ie variable to variable bindings), or as environments for constructed terms (ie environment references in molecules).

This optimization to the basic interpretation algorithm is called **DETERMINISTIC-FRAME OPTIMIZATION (DFO)**, and can result in significant improvement in memory utilization. However, as with all optimization techniques, the effectiveness of DFO is dependent on the Prolog program being interpreted.

As explained earlier, the second constraint is easy to enforce in a NSS system, since it is always possible to orient variable bindings from the more recent to the less recent frame. Constructed terms are recorded on the heap in concrete term instances, making it unnecessary to consult an environment containing the bindings of any variables in the constructed term.

In SS systems, however, it is possible to have variables which outlive the procedure in which they occur. Such variables are referred to as **GLOBAL VARIABLES**. The variable X in Figure 18 is an example of a global variable - its value will be required even after the procedure containing the predicate **a(f(X),X)** as head terminates. The solution to this problem is to use what is called the **TWO-STACK** representation for structure sharing [WAR77][HOG84 pp.205ff][KLU85 pp.176ff].

3.12 The Two-Stack Representation In SS Systems

In a two-stack system, a distinction is made between **LOCAL** (or **PRIVATE**) and **GLOBAL** (or **OUTPUT**) variables. Local variables are those which may be deallocated when a procedure exits since they are not the target of any forward references in earlier environments. Global variables, on the other hand, may need to outlive the procedure in which they occur. The distinction between the two types of variables can be made by a static analysis of the clause during parsing. A variable in a clause is classified as global if some occurrence of that variable in the clause is an argument of a term. If a variable does not occur as a term argument, then it is classified as local. This is because a variable can only outlive its clause instance if some variable in an earlier frame is bound to the term of which it is an argument (note that the inverse is not necessarily true).

The binding environment for a procedure activation is split into two. The environment for the local variables is kept in the frame as before (in a two-stack system, the stack holding the activation frames is called the **LOCAL STACK**), while the environment for the global variables is kept on a second stack called the **GLOBAL STACK.** Since (by definition) the local environment is not the target for forward references, deterministic frames on the local stack may be discarded on procedure exit without creating dangling references. Records on the global stack, however, can only be popped on backtracking.

In a two-stack structure-sharing system, the following constraints are placed on variable binding during unification:

1. both local and global variables may be free or may be bound to a ground (ie variable free) source term,
2. a local variable may be bound to a global variable, or to a local variable in an earlier frame,

3. a local variable may be bound to a constructed term provided the environment component in the molecule representing the constructed term points either to a global environment or to the environment in an earlier local frame, and
4. the binding of a global variable may only refer to global environments.

These constraints do not affect the unification process, while ensuring that deterministic-frame optimization will not result in dangling references being left on the stack.

3.13 Other Optimization Techniques

Deterministic-frame optimization forms the basis for two other optimization techniques, **LAST-CALL-OPTIMIZATION (LCO)** and **TAIL-RECURSION OPTIMIZATION (TRO)**. Both techniques enable the interpreter to overlay the (local) parent frame by the frame of the invoked procedure under certain conditions. The global frame must, of course, be left on the stack until popped by backtracking.

Consider the program fragment

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

Assume the interpreter has just entered procedure b, which is deterministic, and is about to enter procedure d as a result of executing the call in b's body. Since this is the last call in d, and both b and d are deterministic, as soon as d is exited control will return immediately to procedure a, where the call to c is still pending. Consequently, the frame for procedure b can be discarded even before d's frame is created. d's frame is overlaid on b's frame and given a as the parent procedure, thus ensuring that exit from d returns directly to a, as required.

The significance of LCO lies in a refinement called tail-recursion optimization. Recursion is notoriously memory hungry. At the same time, it is the only looping mechanism in Prolog, and hence strategies which minimize the memory demands of recursion are important for the efficient implementation of Prolog. Many Prolog procedures exhibit a particular form of recursion called **TAIL RECURSION** - the last call of the procedure is a recursive call. The usual implementation of the *member* and *append* predicates is a case in point. Like LCO, TRO overlays the frames created by each recursive call in a tail-recursive procedure. With TRO, a tail-recursive procedure requires only a single local frame instead of a frame for each recursive invocation, although a global frame is still required for each such call. This effectively transforms tail recursion into iteration.

3.14 Intelligent Backtracking And Compilation

Other optimization techniques are prompted by considerations of time. Compared with procedural languages, Prolog is not particularly renowned for its execution speed. There are at least two factors contributing to this:

1. Prolog is usually an interpretive language, and
2. the exhaustive search required in finding all possible solutions to a goal.

A Prolog compiler was first suggested and implemented by D.H.D. Warren at the University of Edinburgh [WAR77]. Compilation is to a p-code for an abstract machine (later dubbed **WAM**) supporting the primitives required to realize the interpretation model outlined above. This has made it possible to port the compiler by implementing simulators for WAM (see for example [DEB87]). WAM has also formed the basis for a VLSI implementation of a Prolog processor [CIV89].

Some attempts have also been made to replace Prolog's exhaustive search of the refutation tree by a more intelligent backtracking mechanism [BRU84b][COX84][PER82]. The idea is to have the backtracking system *"learn from previous failures and successes how to get a faster exploration of the remaining alternatives"* [BRU81 p.218]. In such systems, analysis of the unification steps leading to a failure provides information to guide backtracking.

4 IMPLEMENTATION

4.1 Objectives

This chapter describes the implementation of a small structure-sharing interpreter for a subset of the Prolog language (a source listing is given in Appendix B). The interpreter is only meant to demonstrate some implementation principles and to serve as a test-bed for optimization techniques, although the design is sufficiently open to form the kernel of a more practical implementation (though not a full implementation).

4.2 The Prolog Subset

The subset of Prolog selected for this implementation was chosen to be as faithful as possible to logic programming principles. Extralogical features were almost completely excluded from the subset. The principle areas of simplification were:

a. **ARITHMETIC.** Arithmetic is completely excluded from the subset, as are numeric terms.

b. **DATABASE MANAGEMENT**. Dynamic assertion and retraction of clauses is not supported, resulting in a monotonic logic system. Clause manipulation predicates, such as *name* and ..= (univ) are not supported, and would be difficult to implement within the framework of the existing design.

c. **SYNTAX**. The syntax has been kept as simple as possible, with all functors expressed in prefix notation. The only exception is in the representation of lists using the [_|_] notation. The *op* predicate is not supported. This would be rather difficult to implement in the recursive-descent parser used (most Prologs employ an operator-precedence parser). Only pure Horn clauses are allowed - the *or* operator (;) is not supported, although *not* is implemented as a predefined Prolog procedure.

d. **INBUILT PREDICATES**. Only a handful of inbuilt (evaluable) predicates have been implemented, but the mechanism required to support them is already in place, making it a simple matter to extend the list of such predicates.

   The inbuilt predicates currently supported are:

```
!       the cut
nl      the NEWLINE predicate
write   the term display predicate
fail    the predicate which always fails, forcing backtracking.
the lexicographic comparison operators for atoms @>, @<, @>= and
@=<.
```

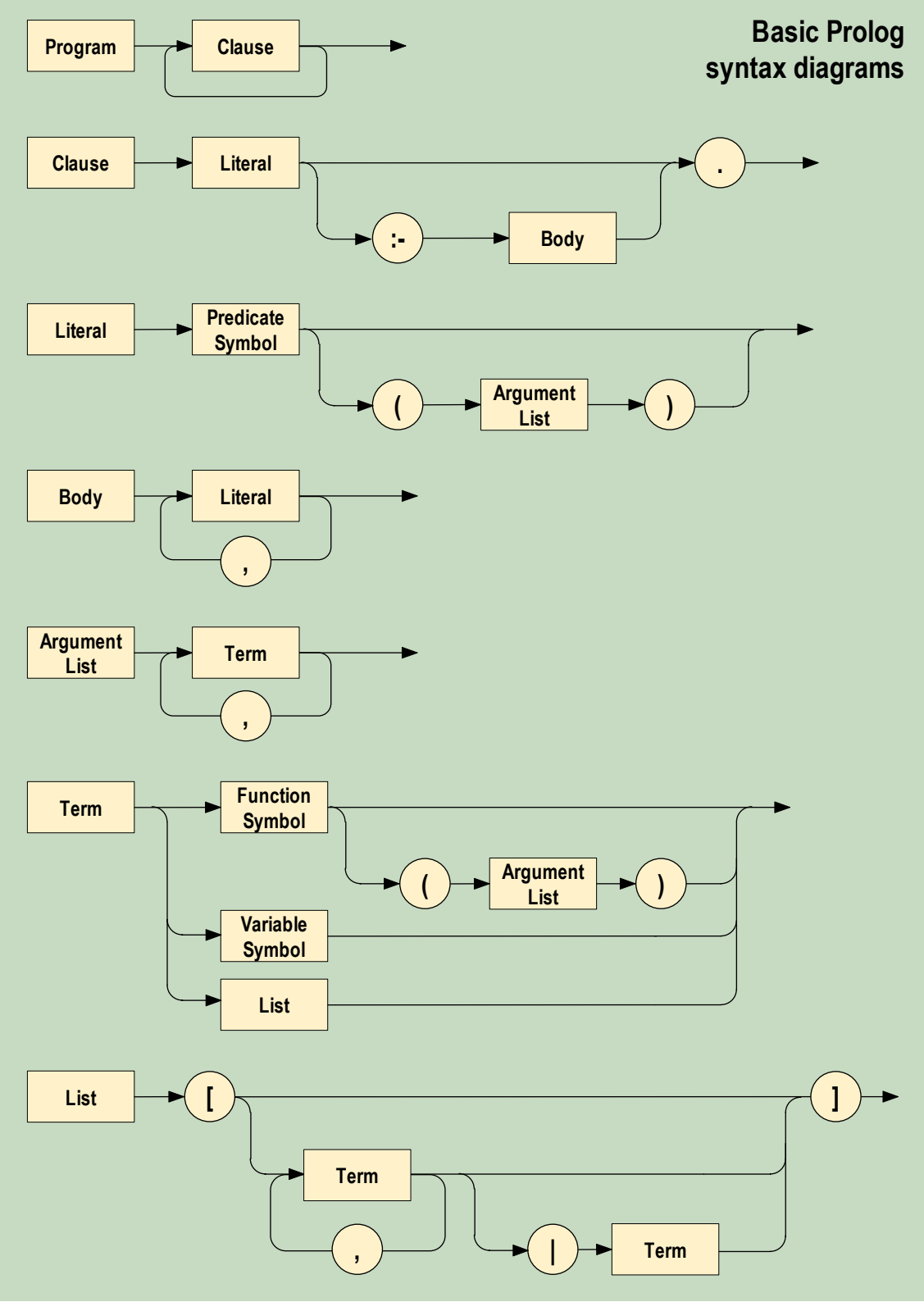
   A few predefined predicates, mostly list manipulation predicates, are implemented in Prolog (see Appendix C).

e. **INPUT/OUTPUT**. Only output (via the *write* inbuilt predicate) is supported. Input is not supported.

The syntax diagrams overleaf define the language accepted by the interpreter.

The interpreter also accepts commands, which are introduced by a period. The commands currently supported are:

```
.LIST  - list database of clauses
.LOAD  - load a file
.STATS - display memory usage statistics
.DEBUG - selectively toggle debugging switches
.STACK - set stack size (in bytes)
.EXIT  - exit interpreter
```



4.3 Choice Of Implementation Language

Various languages were considered for implementing the interpreter. The following features were considered desirable in the implementation language:

1. **MODULARITY**

   support for modular program development, with a clean interface between modules, was required to facilitate experimentation with implementation techniques.

2. **DATA ABSTRACTION AND DATA HIDING**

data abstraction ensures that a program relies only on the data type specification, not on the actual implementation details [MIT88].

3. **LOW-LEVEL SUPPORT**

   in view of the extensive use of dynamic structures in a Prolog interpreter, the prospective language had to offer reasonable access to memory management primitives. Support for address arithmetic was also considered important to facilitate the creation and manipulation of variable-sized records and stacks.

Modula-2 was eventually chosen as the language which offered the best overall balance of all three features. Availability was, of course, another determining factor (Ada would have been a better choice, but was not available). In retrospect, Modula-2 was found to be deficient in the following areas:

1. **PRIVATE and FUNCTION RESULT TYPES.** Although Modula-2 supports private types, these must be declared as pointers to another type declared in the implementation module. Ada-style private types are not supported. Functions are only allowed to return scalar and pointer types. Unconstrained array types are not supported.
2. **POOR STRING SUPPORT.** String support is restricted to a few library procedures. This includes string comparison and concatenation, which are sufficiently elementary to warrant incorporating into the language (as in UCSD Pascal and some versions of Fortran 77).
3. **AWKWARD INPUT-OUTPUT.** Having a separate input and output routine for each different type in the language makes IO both awkward and laborious. A single routine which accepts a variable number of parameters of different types together with formatting information (such as *printf* in C, or even the Pascal *Write*) would have been preferable.
4. **ADDRESS ARITHMETIC.** Address arithmetic has to be performed using library functions. This makes address calculations expensive because of the additional procedure-calling overhead incurred by each calculation.

The version of Modula-2 used for the implementation (JPI version 1.02 for MsDos) has some powerful, although non-standard, library modules. The availability of the library source code, particularly the source code for the memory-management module *Storage*, proved to be of great help in developing the program. On the other hand, the lack of a symbolic debugger made debugging the relatively complicated dynamic data structures quite painful.

Originally it was intended to use *Lex* and *Bison* (the *GNU* implementation of *Yacc*) to generate the scanner and parser for the Prolog interpreter. It is relatively straightforward to modify these two programs to emit Modula-2 instead of the default C or Fortran code. Unfortunately, porting a Unix version of Bison to Microsoft C proved to be more time-consuming than anticipated owing to some hardware dependencies in the code, and the project was eventually shelved.

4.4 Top-Down Design

The implementation comprises 11 primary modules, as follows:

```
VRP:Main module
INBUILT:Definition of inbuilt predicates
PARSE:Top-down parser
LEX:Lexical analyzer
DBASE:database data-structure definitions and handler
STABLE:symbol-table handler
SSTR:string-store handler
COMMAND:command processor
PROCGOAL:the interpreter
STACK:runtime stack and trail handler
STREAMS:low-level input/output
```

The following is a simplified diagram of the hierarchical dependencies between the different modules constituting the interpreter.

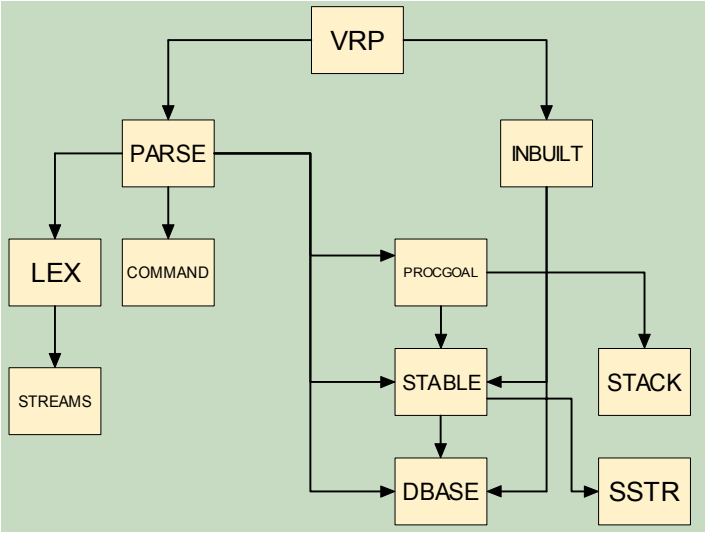


Figure 19: Module hierarchy

The main control loop of the program is shown in Figure 20.

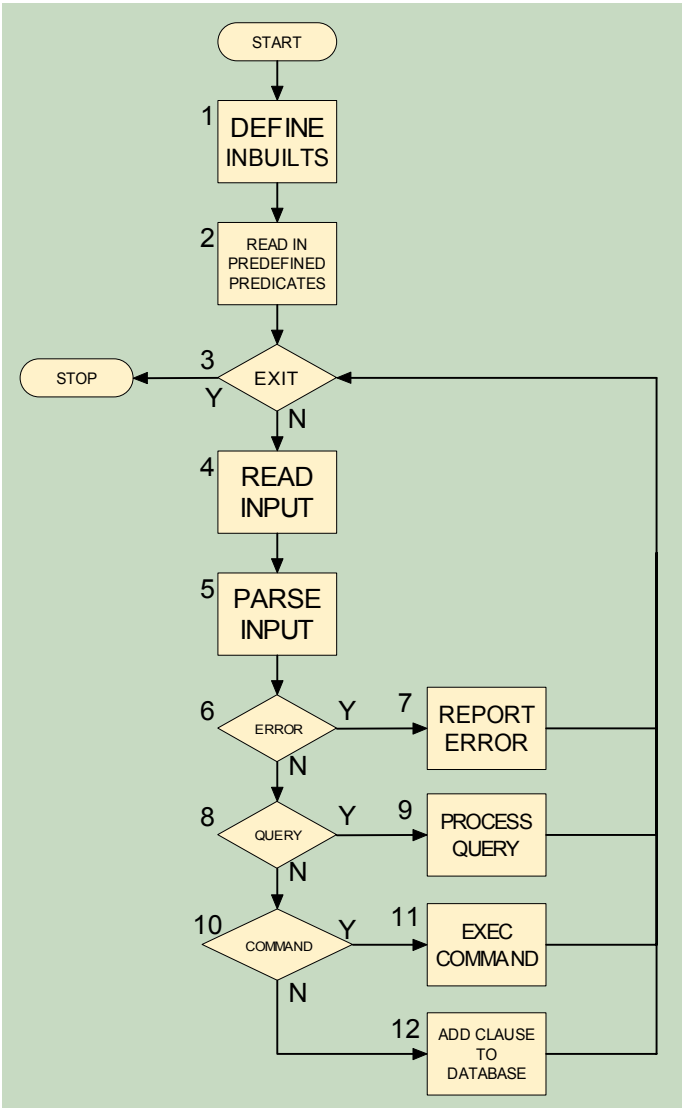


Figure 20: Interpreter main loop

Box Num.	Comments:
1,2	The inbuilt predicates are defined by module Inbuilt, and the predefined predicates are read in from file **PREDEF.PRO**.
3	The interpreter enters a read/parse/process loop. The *Exit* flag is tested at the start of the loop (this flag is set to TRUE when the user issues an **.EXIT** command).
4	Input is read from the current input stream (terminal or file).
5,6,7	The input is parsed by the *Parse* module, which flags any errors.
8	If the input is a query (recognized by the leading :-), then procedure *ProcessGoal* (in module *ProcGoal*) is invoked to execute the query.
9	If the input is a command (recognized by the leading .), then procedure *ProcessCommand* (in module *Command*) is invoked to execute the command.
10	Otherwise the input must be a Prolog definite clause, which is added to the database of clauses.

4.5 The Main Data Structures

The Dictionary

The **dictionary** comprises the string-store, symbol table and internal representation of clauses (the database).

The String Store

The string store maintained by module *SStr* contains the external names of Prolog functors and variables (symbols). Strings passed to the module are stored in a string area, and a string pointer (*Sptr*) is returned to the caller by which the stored string may be referenced.

The *Sptr* is a pointer to an *array[0..MaxStrLen]* of char. The string dereferenced by this pointer is null-terminated so that it can be passed to procedures in the standard *Str* module and to the library *WrStr* procedure, which expect strings in this format.

The module exports two procedures for accessing the string store:

Sstore : stores a string in the string store, returning a pointer of type *Sptr* to the stored string, or **NIL** if insufficient memory remains on the heap.

Sclear : clears the string store and deallocates memory used.

It is up to the caller to impose a structure on the string buffer using the string pointers returned by procedure *Sstore.* The procedure *Sclear* deallocates the string store. The caller must ensure that no dangling pointers remain after a call to *Sclear.*

The string store is implemented as a linked list of string areas (*AreaRec*). Each area contains a 2-field header:

NxtArea : pointer to next area

NxtFree : index to the next free storage position in this area.

The rest of the string area is an array of *AreaSiz* characters. Strings are stored sequentially in the array, and terminated by a null-character. Initially the string store consists of a single empty area. A new area is added to the HEAD of the list when the current one becomes full (ie when the length of the string to be stored exceeds the remaining space).

The structure of the String Store is shown in Figure 21:

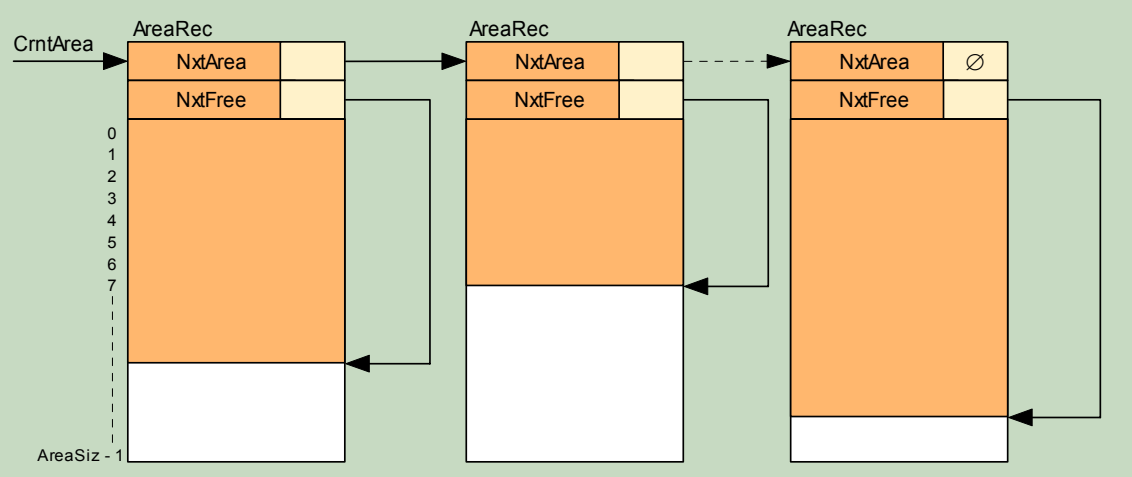


Figure 21: String Store showing linked buffers

The Symbol Table

The symbol-table imposes a structure on the string store and organizes the clause database for fast access during the addition of new clauses. The symbol table is organized using a simple hashing technique based on the first character of symbol names. Although simple, the scheme is sufficient for the purpose required, and has the added benefit of maintaining the predicate symbols sorted in lexicographic order as required by the database listing procedure (exported by module *STable*).

The hashing function maps a symbol onto an entry in an array (*SymTab*, declared in module *STable*) indexed by the first character of the symbol. Each entry in this array is a pointer to a singly-linked list of records of type *SymTabRec*, one for each symbol in the database. Records in each linked list represent symbols having the same initial character in their name. The (simplified) format of the *SymTabRec*, declared in module *DBase*, is :

```
Next    : Pointer to next SymTabRec in this linked list
Name    : Pointer to symbol name string (in string store)
Mode    : symbol mode (see section 4.7)

Count   : for variable encoding (see section 4.11)
          CASE SType : SymType OF
               functor  :
                    Arity   :  Arity of functor
                    FstCls  :  Pointer to head of clause list
                    LstCls  :  Pointer to tail of clause list
             |  variable :
```

In the case of functor symbols the arity is also recorded. Functors with different arity are considered to be distinct symbols and have separate entries. Symbol-table entries for functors also have pointers to the list of clauses (procedure) having this functor as predicate in the clause head. A pointer to the last clause is maintained to facilitate the addition of new clauses to the list.

Within each linked list of the symbol table, records are kept in ascending lexicographic order. In the case of functor records with the same name, the arity is used as a secondary sorting key.

The Clause Records

A clause record (*ClauseRec*, declared in module *DBase*) is kept for each clause in the database. Clause records for the same procedure are kept in a linked list accessed through the symbol-table entry for the predicate in the head of the clause. Clause records are maintained in the order of declaration, as required by the interpretation strategy. The format of a *CluaseRec* is as follows:

```
Next  :  Pointer to next clause
CASE InBlt : BOOLEAN OF
               TRUE  : Proc  : code representing an inbuilt procedure
                       Entry : pointer to symbol-table entry for
                               the name of this inbuilt procedure
             | FALSE : Vars  : Number of variables in this clause
                       Head  : pointer to the head of this clause
                       Body  : pointer to the body of the clause
```

Unlike user-defined and predefined procedures (which are normal Prolog clauses), clauses representing inbuilt procedures are not written in Prolog, and so do not have a head and a body. The *Vars* field is required during interpretation to calculate the amount of stack-space required to accommodate an activation record for this clause. Again, inbuilt procedures do not (in general) require an activation record, and so do not require this field.

The Term Records

The head and body of clauses are represented internally as linked lists of term records (*TermRec*, defined in module *DBase*). *Term* here does not imply that each record represents a term, merely that these structure are used in constructing internal term representations (source terms). The format of a *TermRec* is as follows:

```
Next  :   Pointer to next TermRec in list
Entry :   Pointer to symbol-table entry
CASE SType  : SymType OF
              list,functor : Args : pointer to linked list
                                    of TermRecs representing
                                    arguments if any
            |  variable     : Ofst : variable number within clause
```

The *SType* field flags the record as representing a variable, functor, list constructor or anonymous variable. A record representing the anonymous variable does not require any other field.

If the term record represents a list or a functor, then a linked list of *TermRecs* representing the arguments (if any) is appended. Lists are stored just like any other structure, ie **[a,b,c]** is considered shorthand notation for the structure **.(a,.(b,.c(,[])))**, where . represents the list constructor functor.

In the case of variables, the *Ofst* field records the variable's number within the clause. The parser assigns a number, starting with 0, to each new variable encountered in parsing a clause. This number is used during interpretation as an offset into the variable-binding area (environment vector) of the clause's activation record. The Entry field is only used for records representing functors and variables.

It is necessary to include the symbol type (*SType*) within *TermRec* because, although in the case of variables and functors the type can be read from the symbol-table entry, in the case of special terms such as list constructors and anonymous variables a symbol-table entry does not exist. Also, including the symbol type here makes for less dereferencing when examining terms.

The examples below demonstrate term representation using records of type *TermRec*.

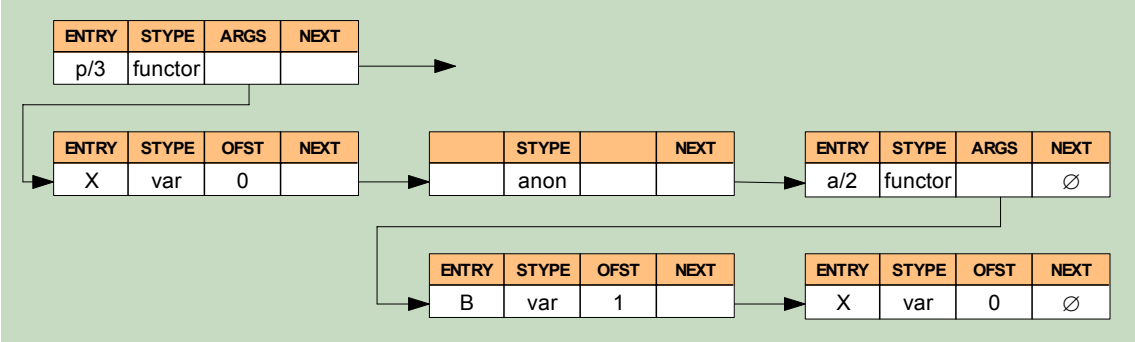


Figure 22: Representation of the term p(X,_,a(B,X))

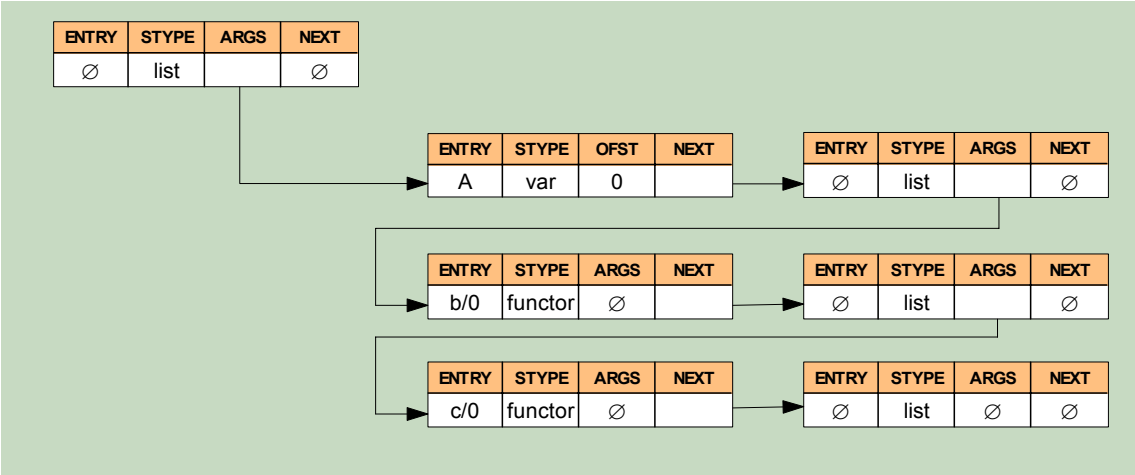


Figure 23: Representation of the term [A,b,c], equivalent to .(A,.(b,.(c,[])))

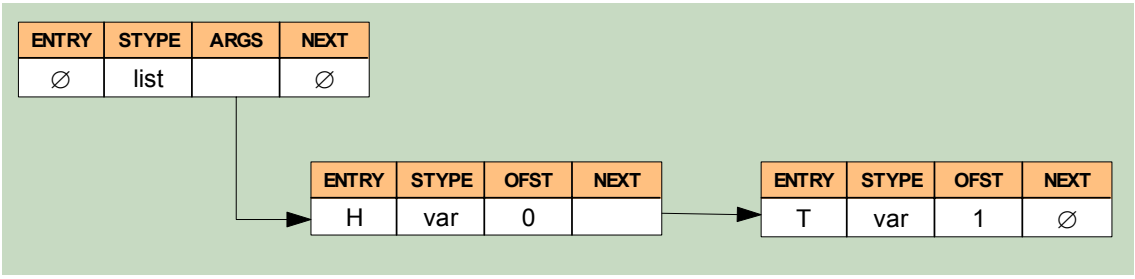


Figure 24: Representation of the term [H|T], equivalent to .(H,T)

Figure 25 shows how a clause is stored in the database.

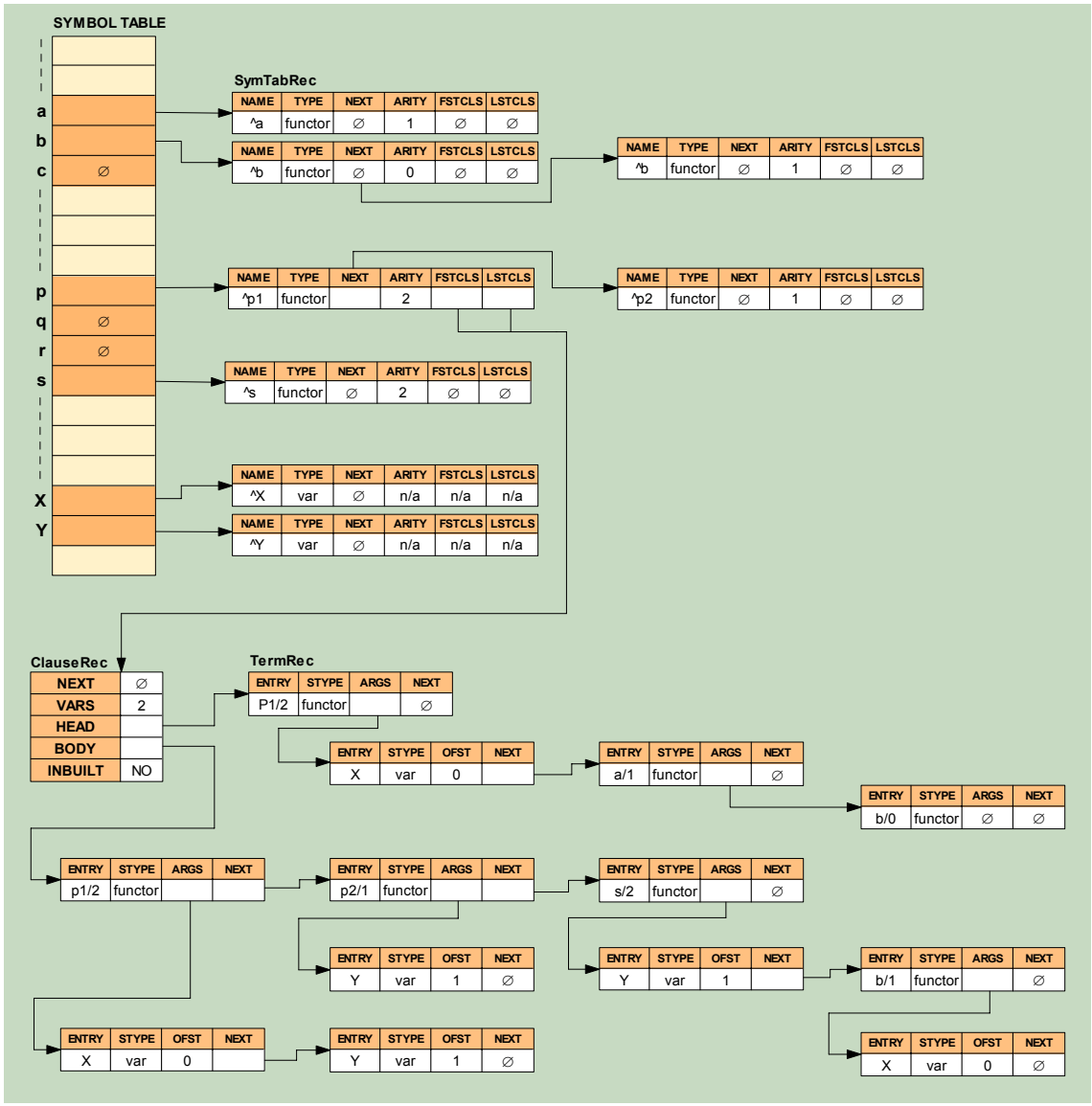


Figure 25: Representation of p1(X,a(b)) :- p1(X,Y), p2(Y), s(Y,b(X)).

The relationship between the string store, symbol table, and database is summarized in Figure 26.

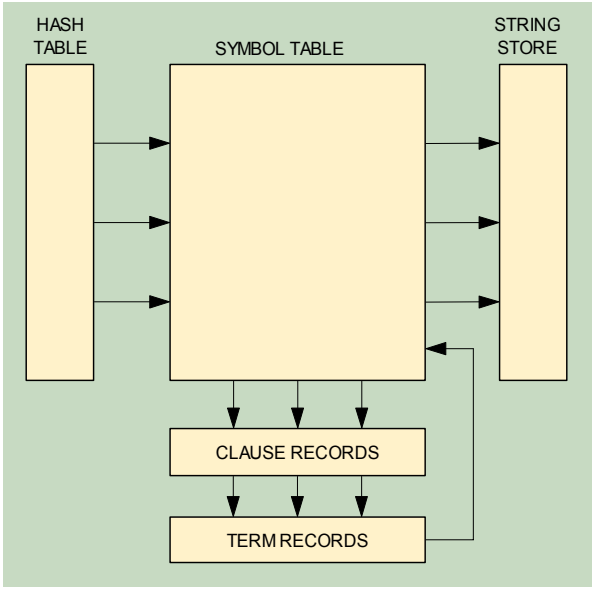


Figure 26: Relationship between symbol table, string store, and database

4.6 Operations On The Symbol Table And Database

The symbol-table module *STable* allows insertion of new symbols into the symbol table by means of procedure *Insert,* which takes as parameters the print name (as a pointer to a string in the string store), type and (if applicable) arity of the symbol and creates a new entry in the symbol table if one doesn't already exist. A pointer to the entry is returned.

No provision presently exists for deleting symbol-table records. Such deletions are not easy to implement, since a check has to be made that no references to the entry to be deleted remain in the database (otherwise dangling-pointer problems may arise).

The symbol-table module is also responsible for producing listings (ie external representations) of terms and clauses in the database.

The database module *DBase* exports various functions and predicates for examining term records, as well as constructors and destructors for records of type *SymTabRec, ClauseRec* and *TermRec.* Note that, since a Prolog database is static in the sense that clauses can be asserted and retracted but not changed, provision for altering database structures is neither provided nor required.

The *DBase* module maintains a free list for each of the tree types of structures required by the database. When a free list becomes exhausted, a chunk of memory is requested from the system and divided into structure-sized units which are linked to form a new free list. This scheme is preferable to invoking the system **ALLOCATE** each time a new structure is to be created because:

1. system storage requests carry a not-insignificant time overhead. This is evident from examination of the code for the standard Storage module, which uses a first-fit algorithm in allocating blocks on the system heap.
2. the Storage module can only allocate memory in paragraph-sized chunks (16-bytes). Had structures to be allocated individually from the system heap, the wastage per unit allocation would be as follows:

Structure	size (bytes)	allocation (paras)	wastage (bytes)
SymTabRec	20	2	12
TermRec	13	1	3
ClauseRec	15	1	1

Given an average of 20 term records, 5 symbol-table records and 1 clause record per clause, this amounts to a hefty 121-byte overhead for each clause in the database.

Constructors for these structures merely fetch the first record on the free list (calling an allocation procedure to create a new free list if this becomes exhausted). Similarly, destructors return a record to the free list where it can be reused. Since the destructors never release memory once allocated (except when the program terminates), system memory may become tied up on the database free lists, resulting in insufficient heap-space for the interpreter stack and trail. Although this is unlikely, a better memory management scheme than the one currently implemented is clearly required. Such a scheme would have to be able to reallocate database records to avoid memory fragmentation. This is not easy, given the number of pointers that will have to be adjusted.

Another advantage of having all requests for structure creation and disposal channelled through the constructor and destructor functions rather than go directly to the system's **ALLOCATE** and **DEALLOCATE** procedures is that it makes collecting memory usage statistics easy. In the course of developing the interpreter, such statistics were found useful in gauging the memory requirements of a typical Prolog program

4.7 Symbol Modes

The interpreter functions in two modes, system and user. A field in the symbol-table record, Mode, is used by the Insert procedure of module *STable* to record the mode under which a symbol has been declared. Initially the interpreter is in system mode. After the inbuilt-predicates have been declared (in module Inbuilt) and the predefined predicates in the file **PREDEF.PRO** loaded, the interpreter switches to user mode and starts processing user input.

Thus all symbol-table entries for inbuilt and predefined symbols have their *Mode* field set to system, while all user entries have their Mode field set to user. The parser uses the *Mode* information to disallow the redefinition of system symbols by the user (see below).

4.8 The Lexical Analyzer

The parsing of clauses is handled by a simple recursive-descent parser implemented in module *Parse.* The procedure which drives the parser, procedure *Reader,* is in fact the main loop of the interpreter, continuously reading in and parsing input from the current input stream (file or terminal).

The input stream is preprocessed by module *Lex,* which fetches tokens on demand and hands them to the parser. A shared structure *CrntTkn* is used to pass tokens from the lexical analyzer to the parser.

A token is a pair consisting of a symbol class and a symbol instance. The class describes the type of token read in, and can be one of:

ColonHyphen	**:–**
Comma	**,**
OpnBrk	(
ClsBrk	)
OpnSqr	[
ClsSqr	]
Bar	|
Dot	**.**
AnonymVar	_
VarSym	an identifier starting with an uppercase
NonVarSym	`an identifier which is not a variable symbol`
FileEnd	`End of file on input`

If the token class is either *VarSym* or *NonVarSym,* the instance field of the token contains the identifier string read in. The instance field is not required for the other token classes (since there is only one instance of each class).

4.9 The Parser

The parser is a straightforward implementation of the following CFG (terminal symbols are printed in bold).

```
<program>       ::=   <clause> { <clause> }
<clause>        ::=   <predicate> [:- <body>] .
<predicate>     ::=   <predicate symbol>  [ ( <argument list> ) ]
<body>          ::=   <literal> { , <literal> }
<literal>       ::=   <variable symbol> |
                      <predicate>
<argument list> ::=   <term> { , <term> }
```

```
<term>          ::=   <variable symbol> |
                      <structure>
<structure>     ::=   <constant symbol> [ ( <argument list> ) ] |
                      <list>
<list>          ::=   [ { <term> { , <term> } [ | <term> ] } ]
<goal>          ::=   :- <body> .
```

with a separate procedure for each non-terminal, as is usual in a recursive-descent parser. Note that, although strictly speaking a variable is NOT a literal, the grammar allows variables to take the place of a literal in the body of a clause. Such a variable represents an indirect procedure call, and must be bound to a 'normal' literal at runtime.

The parser takes advantage of syntactic similarities between certain non-terminals of the language such as <structure> and <predicate>, which are collapsed into a single procedure.

4.10 Constructing The Internal Representation Of A Clause

Parsing a clause (we will for the moment ignore goal clauses) starts with procedure *PrsClause* constructing a *ClauseRec* ready to receive the clause representation. *PrsPred* is then called to parse the predicate at the head of the clause, returning a pointer to a *TermRec* (possibly with linked argument terms) representing the head predicate. If a *ColonHyphen* token is encountered next, *PrsBody* is called to parse the body of the clause, returning a pointer to a linked list of *TermRecs* representing the literals in the body of the clause. Otherwise the *Body* field of the *ClauseRec* is set to **NIL.**

The main work of parsing the head and body of a clause, and of creating the *TermRecs* to represent these objects, is carried out by procedures *PrsLiteral* (with support procedure *PrsArgList*) and *PrsTerm* (with support procedure *PrsList*). Each of these procedures returns (on successful termination), a pointer to a *TermRec,* which is linked to the *TermRec* created by the caller. This way, the parser constructs the linked lists of *TermRecs* representing the terms, literals and predicates in a clause.

The following pseudo-code summarizes the process of parsing a clause. We assume that *Token* always contains the next token to be parsed (in reality, this has to be fetched from the lexical analyzer with a call to *GetToken*). For simplicity, *PrsList* is omitted. We let *MakeTerm* be a constructor of *TermRecs,* returning a pointer to a new instance of a term record. Similarly *MakeClause.* The notation **Field(Pointer)** is used instead of the more cumbersome **Pointer^.Field**.

```
PROCEDURE PrsClause
   C := MakeClause;
   Head(C) := PrsPred;
   IF Token = ColonHyphen THEN
      Body(C) := PrsBody
   ELSE
      Body(C) := NIL;
   link C into the database
END PrsClause;

PROCEDURE PrsPred : TermRecPtr;
   RETURN PrsLiteral;
END PrsPred;

PROCEDURE PrsBody : TermRecPtr;
   First := T := PrsLiteral;
   WHILE Token = Comma DO
      Next(T) := PrsLiteral;
      T := Next(T);
   RETURN First;
END PrsBody;
```

```
PROCEDURE PrsLiteral : TermRec;
   CASE Token OF
      NonVarSym : T := MakeTerm;
                  IF Token = OpnBrk THEN
                     Args(T) := PrsArgList;
                  RETURN T;
      VarSym  : RETURN PrsTerm;
END PrsLiteral;

PROCEDURE PrsArgList : TermRec;
   First := T := PrsTerm;
   WHILE Token = Comma DO
      Next(T) := PrsTerm;
      T := Next(T);
   RETURN First;
END PrsArgList;


PROCEDURE PrsTerm : TermRec;
CASE Token OF
   VarSym    : RETURN MakeTerm;
   NonVarSym : RETURN PrsLiteral;
   OpnBrk    : RETURN PrsArgList;
END PrsTerm;
```

When a new *TermRec* is created, the details of the term just parsed (primarily its type and a pointer to the symbol-table entry of its functor or variable identifer) are filled in, but this is not shown in the pseudo code.

Linking a clause into the database involves locating the symbol-table record for the predicate at the head of the clause. The clause is then linked following the clause pointed to by the *LstCls* field of the symbol-table record, which is updated to point to the new clause. If no clauses yet exist for this predicate (*FstCls*=**NIL**), the *FstCls* and *LstCls* fields are set to point to the new clause.

4.11 Encoding Variables

The encoding of the variables in a clause is performed as the clause is being parsed. A numeric field (*Count*) in the symbol-table record *SymTabRec* is used for this purpose, together with a variable *NVars.*

Count is set to a distinguished value *NoCount* during the creation of a new symbol-table entry, while *NVars* is reset to 0 at the beginning of parsing each clause. When a variable term is encountered by *PrsTerm,* it looks at the *Count* field of the symbol-table entry for that variable symbol. If the *Count* field is still *NoCount* it is set to the current value of *NVars,* which is then incremented. The value of the *Count* field in the symbol-table entry of a variable is copied to the *Ofst* field of the *TermRec.* This way, different occurrences of the same variable within a clause are encoded with the same offset value. The following revised version of *PrsTerm* includes the extra processing required to encode variables:

```
PROCEDURE PrsTerm : TermRec;
   CASE Token OF
      VarSym    : T := MakeTerm;
                  S := Symbol Table Entry of Varsym;
                  IF Count(S) = NoCount THEN
                     Count(S) := NVars;
                     INC(NVars);
                  Ofst(T) := Count(S);
                  RETURN T;
      NonVarSym : RETURN PrsLiteral;
      OpnBrk    : RETURN PrsList;
END PrsTerm;
```

When a clause has been successfully parsed, the final value of *NVars* is the number of distinct variables in the clause, and is recorded in the *Vars* field of the *ClauseRec.* Finally, all symbol table entries of variables within the clause have their *Count* field reset to *NoCount* in preparation for parsing the next clause.

4.12 Preventing Redefinition Of System Predicates By The User

PrsClause prevents the user from redefining system predicates. When the head of a clause has been parsed (by *PrsPred*), *PrsCls* checks to see whether the symbol-table record for the predicate symbol in the head has its *Mode* field set to **SYSTEM.** If so, and the interpreter is currently in user mode, then an error condition is raised and the current clause is abandoned. Note that this does not prevent the user from calling system predicates, since such calls appear as literals in the body of a clause, not as the predicate in the head.

4.13 Parsing Goal Clauses

PrsClause recognizes a goal clause by the first token, which is always *ColonHyphen.* In the case of a goal clause, the head field is set to **NIL,** and the body is parsed by *PrsBody* as for a definite clause. When the goal clause has been successfully parsed, it is not linked into the database, but is passed to procedure *ProcessGoal* (in module *ProcGoal*) for immediate execution.

4.14 The Runtime Structures

Activation Frames

A procedure activation frame is a variable-length record (declared in module Stack) as follows:

```
Frame = RECORD
            Prev       : pointer to start of previous frame
            Parent     : pointer to parent frame
            CrntLit    : pointer to current call
            NxtClause  : pointer to next untried candidate clause
            CrntBTP    : pointer to frame to backtrack to
            Trail      : trail pointer
            Vars       : size of binding array
            Binds      : variable-sized array of Binding records
        END;
```

The first field of the frame contains a pointer to the start of the previous frame on the stack, and is only required by the stack-handling routines (discussed below). The next five fields hold control information required by the interpreter.

The *Vars* field records the number of variables in the activated clause, which is also the number of entries in the bindings array *Binds.* Each variable in the clause is mapped onto an entry in this array by the offset number (*Ofst*) assigned during parsing. The bindings array records the environment (ie variable assignments) defining a clause instance.

Since Modula-2 does not support dynamically-dimensioned arrays, *Binds* is actually declared as an array of size *MaxVars* (a constant defined in module *DBase,* declaring the maximum number of distinct variables allowed in a single clause - currently 1000). However, the frame-creation routine *MAKEFrame* (in module *Stack*) only allocates memory for *Vars* entries in the array, so that the *Vars* field completely determines the size of a frame.

Binding Records

Each entry in array *Binds* records the binding of a single variable in the clause instance represented by the frame. The binding of a variable encoded with an offset number x in a clause is recorded in the entry *Binds[x]* of the frame representing an instance of the clause. Three types of bindings are recognized by the interpreter:

```
free -  if a variable is not currently bound
lit  -  if the variable is bound to a term skeleton
var  -  if the variable is currently bound to another variable.
```

The following schematic (Figure 27) illustrates these three types of bindings.

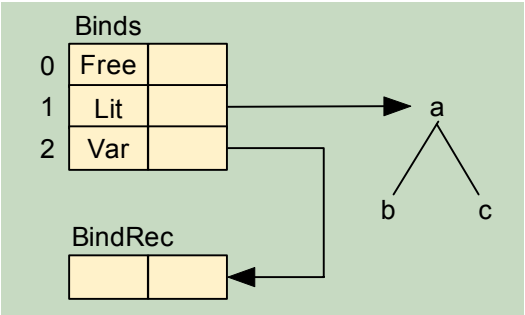


Figure 27: Three different variable bindings

Each variable binding is recorded in a record of type *Binding,* declared as:

```
Binding  = RECORD
           CASE BType : (free, var, lit) OF
                var   : BPtr : ptr to another binding record
             | lit   : TPtr : ptr to a term skeleton
                       Env  : ptr to frame containing environment
           END;
           END;
```

If the variable is bound to another variable, then the field *BPtr* in its *Binding* record contains a pointer to the binding record of the variable to which it is currently bound.

If the variable is bound to a term, then field *TPtr* contains a pointer to the term-skeleton in the database, while field *Env* contains a pointer to a frame containing the environment for the variables in the term skeleton. For example, assume that variable 0 in a clause is bound to the term **a(X:1)**, where the notation **X:1** represents variable X encoded as offset 1. The following diagram shows the situation, with frame 1 providing the environment for the interpretation of variable X in the term **a(X:1)** to which variable 0 of frame 2 is bound. In this case, X is a free variable.

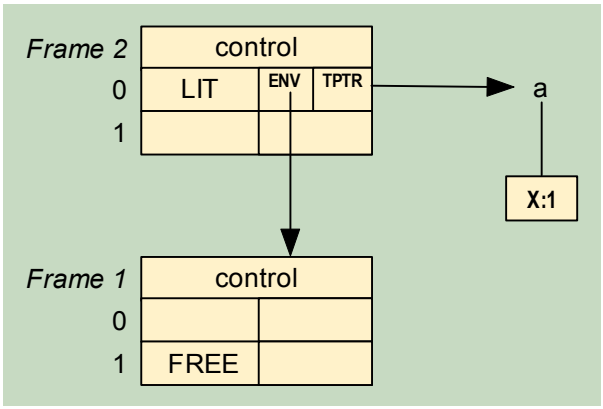


Figure 28: Variable 0 in Frame 2 is bound to the term a(X:1), with Frame 1 as environment

The Stack Module

The stack module *Stack* manages the runtime stacks - the activation (frame) stack and the trail. The frame stack and the trail share a common block of memory and grow towards each other from opposite ends of the block.

While trail records are of uniform size (consisting merely of pointers to binding records), the procedure activation frames held on the runtime stack are variable-sized. Because of this, and because a temporary frame is needed by the unification process, the two stacks are handled very differently. In particular, *TRAILTOP* points to the NEXT free location on the trail, while

STACKTOP points to the LAST OCCUPIED location on the stack. Also, frame records require a pointer to the previous record on the stack.

The main pointers associated with the two stacks are:

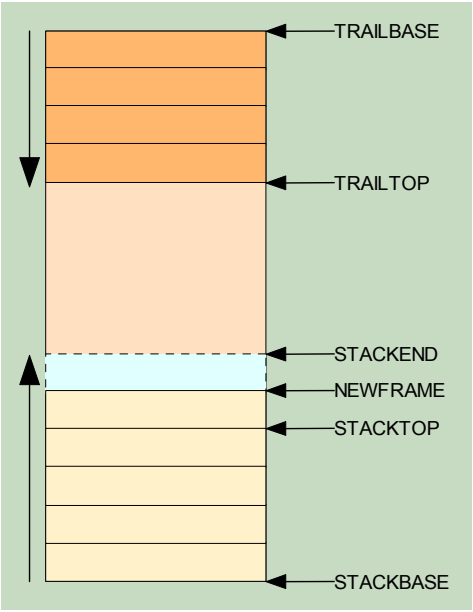


Figure 29: The runtime stacks.

NewFrame is required by the unification procedure. At the start of a unification, a call to *MAKEFrame* creates a *NewFrame* of the required size just beneath *StackTop.* If the unification succeeds, *NewFrame* is pushed onto the stack and becomes the new *StackTop.* Otherwise it is overwritten by the subsequent creation of a new *NewFrame. StackEnd* keeps track of the location where *NewFrame* ends, and is required in checking for collisions between the procedure-activation stack and the trail.

Besides exporting the frame and binding-record types, module *Stack* also provides frame creation and stack operations. The main stack operations are:

MAKEFrame (Vars) : Frame Pointer

Returns a pointer to a new stack frame immediately below stacktop large enough to accommodate *Vars* binding records. This is the frame that gets pushed next time *PUSHFrame* is called. The *Vars* and *Trail* fields of the new frame are initialized, and all variable bindings are set to free. **NIL** is returned if creating the frame would result in the frame stack colliding with the trail.

PUSHFrame

Pushes the frame created by *MAKEFrame* (*NewFrame*) onto the stack. This entails setting the *Prev* field of *NewFrame* to the current value of *StackTop,* setting *StackTop* to *NewFrame,* and *NewFrame* to *StackEnd.*

POPFrames (Frame Pointer)

Pops all frames on stack from and including the frame pointed to by Frame Pointer. All variables instantiated from this frame onwards are uninstantiated, and *TrailTop* is reset to the value recorded in the frame referenced by Frame Pointer.

Module *Stack* also provides routines which record variable instantiations on the Trail (*STORETrail*), dereference variables bound to other variables (*DeRef*), return the address of the binding record of a variable within a frame (*BindAdr*) and test whether a variable is free or instantiated.

4.15 The Interpreter

The interpreter starts by creating a frame representing entry into the goal, which is made the current procedure. Processing then proceeds as shown in Figure 30. The following comments refer to the numbered boxes in the figure.

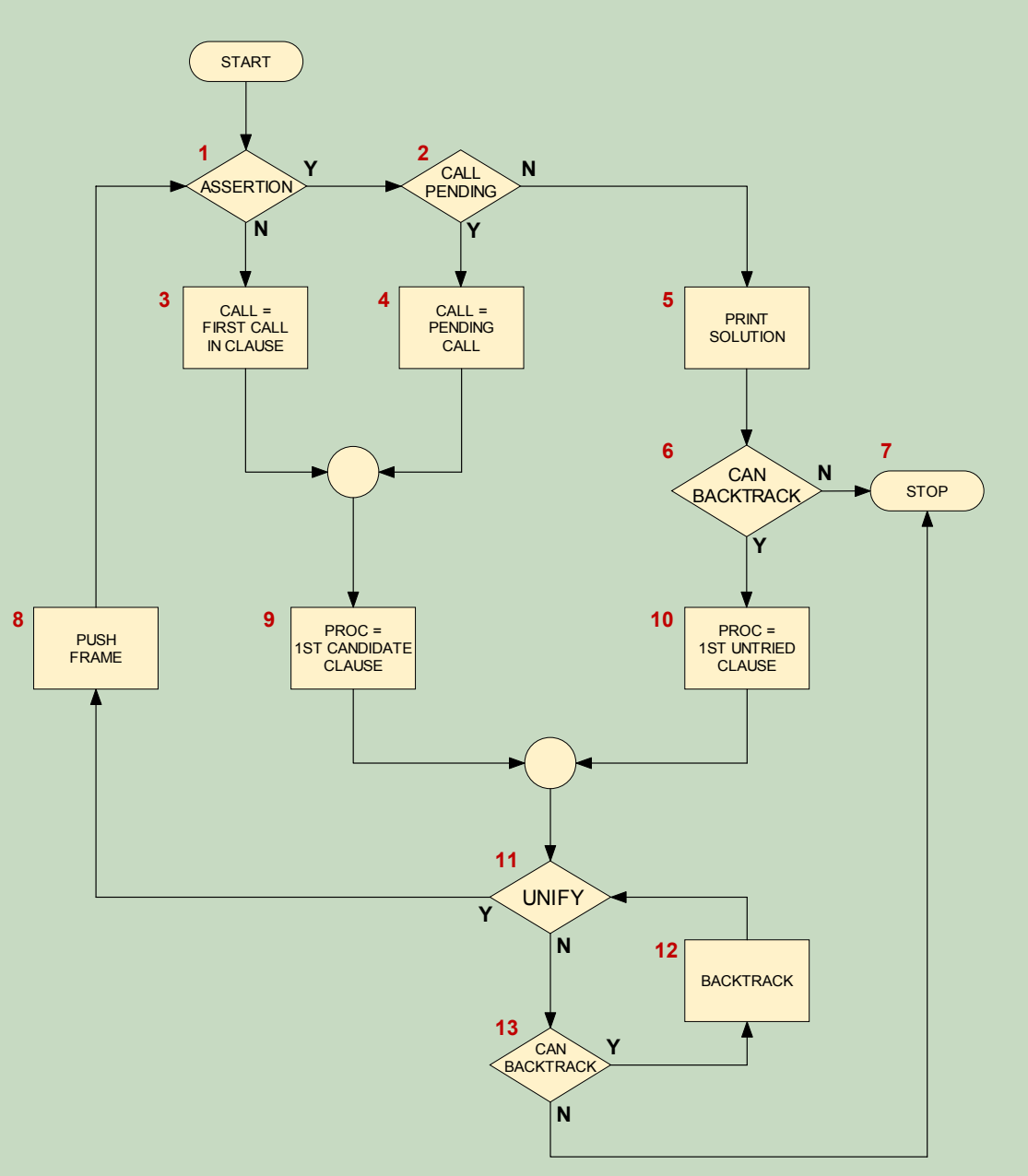


Figure 30: Simplified interpreter flowchart.

Box Num.	Comments:
1	If the current clause is an assertion (ie has no body), then a PROCEDURE EXIT step is immediately executed, and the parent procedure is resumed at its next call (boxes 2 and 4). Otherwise, a PROCEDURE ENTRY step is executed (box 3), with the first call in the body of the procedure being the new CALL.
2	On procedure exit, the frame stack is searched backwards looking for a parent procedure which still has some pending calls. If no pending calls are left, then a solution has been found, and the bindings of variables in the goal clause (if any) are printed out (box 5).
3,4	**CALL SELECTION**. The next call selected is either the first call in the current

procedure (if this is not an assertion), or the next call of the most recent parent procedure which still has some pending calls left.

5,6 After printing out a solution, the interpreter attempts to backtrack in search of further solutions. Backtracking involves popping all frames up to and including the most recent backtrack-point and trying a different path in the search tree. If there is no backtrack-point, then backtracking is not possible, so execution terminates.

9,10 **PROCEDURE SELECTION**. The next candidate clause to be tried is either the first clause for the current call if no backtracking has occurred (box 9) or, in the case of backtracking, the next untried clause for the call backtracked to (box 10).

11 **UNIFICATION**. An attempt is made to unify the current call with the head of the current (candidate) clause. A record of the variables instantiated in the course of unifying the call and the head of the candidate clause is kept on the trail. Unification constructs a temporary frame for the candidate clause (with a call to *MAKEFrame*), which is pushed onto the stack (box 8) if the unification succeeds, but is otherwise discarded.

12,13 If unification fails, then the interpreter attempts to backtrack. Shallow backtracking - the process of trying a different candidate clause for the current call - is attempted first. If this fails, then deep backtracking - backtracking to a previous call - is attempted.

8 If unification succeeds, then the temporary frame constructed during the unification process is pushed onto the stack. The clause whose head successfully unified with the current call becomes the procedure to be entered next. If some untried candidate clauses remain for this call, then the frame just stacked becomes the most recent backtrack-point.

When a call to an inbuilt procedure is encountered following the call selection process in boxes 3 and 4, the Proc field of the *ClauseRec* representing the inbuilt procedure is used to determine the action to be performed. No frame is constructed for a call to an inbuilt procedure, and interpretation passes immediately to the next call in the body of the current procedure.

A count is kept of the number of solutions output by box 5. If the interpretation procedure terminates (box 7) without having found any solutions, then the query fails, and the message NO is output.

4.16 Unification

The unification procedure *UnifyTerm* (in module *ProcGoal*) takes two term instances as parameters and attempts to unify them, recursively traversing the term trees to unify the terms' arguments. The procedure succeeds (returning TRUE) if the terms are unifiable, instantiating variables in the process. Otherwise the procedure fails, returning FALSE.

Each term instance consists of a pointer to a term skeleton in the database, and a pointer to a frame containing the environment. This is the usual representation of term instances in a structure-sharing Prolog implementation. The following pseudo-code sketches the unification algorithm. We assume that the two terms T1 and T2 are completely dereferenced - ie if either term is a variable, then it is replaced by the term to which it is bound (which may be a free variable).

```
PROCEDURE UnifyTerm (T1,T2 : TermPtr; E1,E2 : FramePtr) : BOOLEAN;
IF either term is the anonymous variable THEN RETURN TRUE;

IF neither term is a free variable THEN
   IF T1 and T2 do not have the same functor THEN RETURN FALSE
   ELSE FOR each argument A1,A2 of T1,T2 DO
      IF NOT UnifyTerm (A1,A2,E1,E2) THEN RETURN FALSE
   RETURN TRUE;

At this point, at least one of T1,T2 must be a free variable. This
leads to the following cases:

1.  Both T1 and T2 are free variables:
    Bind T2 to T1

2.  T2 is free, while T1 is a structure or a variable bound to
    a structure.
    Bind T2 to T1 with E1 as environment.

3.  T1 is free, while T2 is a structure or a variable
    bound to a structure.
    Bind T1 to T2 with E2 as environment.

   RETURN TRUE;
END UnifyTerm;
```

Binding a free variable T_i (encoded as offset O_i) to a term T_j involves altering the binding record indexed by O_i in the binding array of frame E_i. If T_j is a structure (or a variable instantiated to a structure), then the binding record for T_i is made to point to the structure T_j, with E_j as environment for any variables in T_j. If T_j is a free variable, then T_i is made to point to the binding record of T_j within frame E_j.

When a variable is instantiated during unification, a pointer to the variable's binding record is pushed on the trail so that all bindings can be undone on backtracking. Since backtracking pops all frames created since the last backtrack-point, only instantiations of variables in frames earlier than the latest backtrack-point need be recorded.

The algorithm does not perform the occur check, as is normal in Prolog. This may lead to the construction of cyclic structures, but makes the algorithm linear in the number of subterms for the two terms to be unified. The occur check was eliminated from the original Marseille interpreter [COH88] for pragmatic reasons. In a way, this simplification of Robinson's unification algorithm is what has made Prolog a viable programming language. Some implementers have taken advantage of the potentially cyclic structures generated by the simplified unification algorithm, and allow the construction and handling of infinite terms in Prolog (see [FIL84][HAR84]).

4.17 Enhancements To The Interpreter

Besides the implementation of various optimization techniques, primarily deterministic-frame optimization (which could form the basis for implementing both LAST-CALL OPTIMIZATION and TAIL-RECURSION OPTIMIZATION), enhancements may include the addition of arithmetic and a better parser.

The recursive-descent parser used is neither very efficient nor suitable for Prolog. Problems would be encountered in implementing predicates like *op* and *clause*, which are easier to implement in a bottom-up parser.

The representation of terms may also be improved - a more compact representation of lists is possible, although this will somewhat complicate the unification algorithm. Memory management also needs improving to control the amount of memory which currently accumulates on the freelists maintained by the database module. This would require reallocating

clauses in the database, which is not a straightforward matter since a potentially large number of pointers may need to be adjusted.

BIBLIOGRAPHY

KEY:
ACM TOPLAS *ACM Transactions on Programming Languages and Systems*
CACM *Communications of the ACM*
CM *Contemporary Mathematics*
JACM *Journal of the ACM*
JLP *Journal of Logic Programming*
LNM *Lecture Notes in Mathematics*

[AND76] Andreka,H. and Nemeti,I. (1976) **The Genralized Completeness of Horn Predicate Logic as a Programming Language**, DAI Report No.21, University of Edinburgh.

[APT82] Apt,K.R. and van Emden,M.H. (1982) *Contributions to the Theory of Logic Programming*, in **JACM** Vol.29/3 pp.841-862.

[BOW82] Bowen,K.A. (1982) *Programming with Full First-Order Logic*, in [HAY82].

[BRU81] Bruynooghe,M. (1981) Intelligent Backtracking for An Interpreter of Horn Clause Logic Programs in [DOM81].

[BRU82] Bruynooghe,M. (1982) The Memory Management of PROLOG Implementations, in [CLA82].

[BRU84a] Bruynooghe,M. (1984) *Garbage Collection in Prolog Interpreters*, in [CAM84].

[BRU84b] Bruynooghe,M. and Pereira,L.M. (1984) *Deduction Revision by Intelligent Backtracking*, in [CAM84].

[CAM84] Campbell,J.A. (1984,ed.) **Implementations of Prolog**, Ellis Horwood.

[CHA74] Chang,C. and Lee,R. (1974) **Symbolic Logic and Mechanical Theorem Proving**, Academic Press.

[CIV89] Civera,P., Piccinini,G., Zamboni,M. (1989) *Implementation Studies for a VLSI Prolog Coprocessor*, in **IEEE Micro**, Vol.9/1 February 1989 pp.10-23.

[CLA82] Clark,K.L. and Tarnlund,S.-Å. (1982,eds.), **Logic Programming**, Academic Press, London.

[CLO81] Clocksin,W.F. and Mellish,C.S. (1981) **Programming in Prolog**, Springer Verlag.

[COH88] Cohen,J. (1988) *A View of the Origins and Development of Prolog*, in **CACM** Vol.31/1 pp.26-36.

[CON89] Conlon,T. (1989) **Programming in Parlog**, Addison-Wesley.

[COX84] Cox,P.T. (1984) Finding Backtrack Points for Intelligent Backtracking, in [CAM84].

[DEB87] Debray,S.K. (1987) **The SB-Prolog System, Version 2.2**, Department of Computer Science, University of Arizona.

[DEB89a] Debray,S.K. (1989) *Static Inference of Modes and Data Dependencies in Logic Programs* in **ACM TOPLAS**, Vol.11/3 July 1989 pp.418-450.

[DEB89b] Debray,S.K. and Warren,D.S. (1989) *Functional Computations in Logic Programs*, in **ACM TOPLAS** Vol.11/3 July 1989 pp.451-481.

[DOM81] Domoki,B. and Gergely,T. (1981 eds) **Proc. of Colloquium on Mathematical Logic in Programming 1978, Salgotarjan, Hungary**. Republished by North-Holland Publ. Amsterdam.

[DOW84] Dowling,W. and Gallier,J.H. (1984) *Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae*, in **JLP** No.3 pp.267-284.

[DOW86] Dowsing,R.D., Rayward-Smith,V.J. and Walter,C.D. (1986) **A First Course in Formal Logic and its Applications in Computer Science**, Blackwell Scientific Publications.

[EMD76] van Emden,M.H. and Kowalski,R. (1976) *The Semantics of predicate Logic as a Programming Language*, in **JACM** Vol.23/4 pp.733-742.

[EMD84] van Emden,M.H. (1984) *An Interpreting Algorithm for Prolog Programs*, in [CAM84].

[FIL84] Filgueiras,M. (1984) *A Prolog Interpreter working with Infinite Terms*, in [CAM84].

[GAL87] Gallier,J.H. (1987) **Logic for Computer Science: Foundations of Automatic Theorem Proving**, John Wiley.

[GOO85] Goodman,S.E. and Hedetniemi,S.T. (1985) **Introduction to the Design and Analysis of Algorithms** (2nd printing) McGraw Hill International Student Edition.

[HAR84] Haridi,S. and Sahlin,D. (1984) *Efficient Implementation of Unification of Cyclic Structures*, in [CAM84].

[HAY82] Hayes,J.E. et al (1982,eds.) **Machine Intelligence 10**, Ellis Horwood.

[HOG84] Hogger,J.C. (1984) **Introduction to Prolog Programming**, Academic Press.

[KLU85] Kluzniak,F. and Szpakowicz,S. (1985) **Prolog for Programmers**, Academic Press.

[KNU73] Knuth,D. (1973 2ed) **The Art of Computer Programming Vol.1**, Addison-Wesley.

[KOW88] Kowalski,R. (1988) *The Early Years of Logic Programming*, in **CACM**, Vol.31/1 pp.38-43.

[LOV84] Loveland,D.W. (1984) *Automated Theorem Proving: A Quarter Century Review* in **CM** Vol.29 pp 1-42.

[LUK70] Lukham,D. (1970) *Refinement Theorems in Resolution Theory* in **LNM** Vol.125 pp 163-190.

[MEL82] Mellish,C.S. (1982) *An Alternative to Structure Sharing in the Implementation of a PROLOG Interpreter*, DAI Research Paper No.150, University of Edinburgh. (An abridged version appears under the same title in [CLA82]).

[MIT88] Mitchell,J.C. and Plotkin,G.D. (1988) *Abstract Types have Existential Type* in **ACM TOPLAS** Vol.10/3 pp.470-502.

[NIC89] Nicholson,T and Foo,N. (1989) *A Denotational Semantics for Prolog* in **ACM TOPLAS** Vol.11/4 October 1989 pp.650-665.

[NIL84] Nilsson, J.F. (1984) *Formal Vienna-Definition-Method Models of Prolog* in [CAM84].

[PER82] Pereira,L.M. and Porto,A. (1982) *Selective Backtracking*, in [CLA82].

[RIN88] Ringwood,G.A. (1988) *Parlog86 and the Dining Logicians*, in **CACM** Vol.31/1 pp.10-25.

[ROB65] Robinson,J.A. (1965) *A Machine-oriented Logic Based on the Resolution Principle*, in **JACM** Vol.12/1 pp.23-41.

[SMU68] Smullyan,R.M. (1968) **First-Order Logic** Springer-Verlag.

[THA88] Thayse,A. (1988,ed.) **From Standard Logic to Logic Programming**, John Wiley.

[TIC89] Tick,E. (1989) *Comparing Two Parallel Logic-Programming Architectures*, in **IEEE Software** Vol.6/4, pp.71-80.

[WAR77] Warren,D.H.D. (1977) **Implementing Prolog - Compiling Predicate Logic Programs**, Department of A.I. University of Edinburgh, Research Report No39/40.

APPENDIX A
PROPOS SOURCE CODE

```
DEFINITION MODULE Dbase;

IMPORT Str;

CONST
    maxsymln =  30;

TYPE
    symbol   =  ARRAY [1..maxsymln] OF CHAR;

    HeadPtr  =  POINTER TO HeadRec;
    BodyPtr  =  POINTER TO BodyRec;
    AtomPtr  =  POINTER TO AtomRec;

    HeadRec  =  RECORD
                  sym    : symbol;
                  nxt    : HeadPtr;
                  clause : BodyPtr;
                END;

    BodyRec  =  RECORD
                  nxt    : BodyPtr;
                  first  : AtomPtr;
                END;

    AtomRec  =  RECORD
                  sym    : HeadPtr;
                  nxt    : AtomPtr;
                END;

VAR goal : HeadPtr;

PROCEDURE InsertSymbol (s:symbol) : HeadPtr;

PROCEDURE listing;
PROCEDURE ListClauses (h : HeadPtr);

PROCEDURE NewHead () : HeadPtr;
PROCEDURE NewBody () : BodyPtr;
PROCEDURE NewAtom () : AtomPtr;

PROCEDURE EqStr(s1,s2: ARRAY OF CHAR) : BOOLEAN;

PROCEDURE DisposeClause(H : HeadPtr);

END Dbase.
```

```
IMPLEMENTATION MODULE Dbase;

FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM  IMPORT TSIZE;
FROM IO      IMPORT WrStr, WrLn, WrChar;
FROM Str     IMPORT Length;
FROM AsmLib  IMPORT CompareStr;

TYPE dbase    =  ARRAY ['a'..'z'] OF HeadPtr;
VAR  clauses : dbase;


(* Compare two strings for equality *)

        PROCEDURE EqStr(s1,s2: ARRAY OF CHAR) : BOOLEAN;
        BEGIN
           RETURN (CompareStr(s1,s2)=0);
        END EqStr;


(* Allocate size bytes on the heap, returning pointer *)

        PROCEDURE new (size : CARDINAL) : ADDRESS;
        VAR addr : ADDRESS;
        BEGIN
           ALLOCATE(addr,size);
           RETURN addr;
        END new;


(* Create a new HeadRec *)

        PROCEDURE NewHead () : HeadPtr;
        BEGIN
           RETURN (new(TSIZE(HeadRec)));
        END NewHead;


(* Create a new BodyRec *)

        PROCEDURE NewBody () : BodyPtr;
        BEGIN
           RETURN (new(TSIZE(BodyRec)));
        END NewBody;


(* Create a new AtomRec *)

        PROCEDURE NewAtom () : AtomPtr;
        BEGIN
           RETURN (new(TSIZE(AtomRec)));
        END NewAtom;


(* Dispose of all clauses associated with a HeadRec *)

        PROCEDURE DisposeClause (H : HeadPtr);
        VAR b,b1 : BodyPtr;
            a,a1 : AtomPtr;
        BEGIN
           b := H^.clause;
           WHILE b # NIL DO
              a := b^.first;
              WHILE a# NIL DO
                 a1 := a^.nxt;
                 DEALLOCATE(a,TSIZE(AtomRec));
                 a := a1;
              END;
              b1 := b^.nxt;
              DEALLOCATE(b,TSIZE(BodyRec));
              b := b1;
```

```
            END;
            H^.clause := NIL;
         END DisposeClause;


(* Create and initialize a new HeadRec *)

         PROCEDURE makenode(s:symbol) : HeadPtr;
         VAR p : HeadPtr;
         BEGIN
            p := NewHead();
            p^.sym := s;
            p^.nxt := NIL;
            p^.clause := NIL;
            RETURN p;
         END makenode;


(* Search for a symbol in the symbol table, creating a new HeadRec
   for the symbol if not found. Returns pointer to found/created
   symbol table node *)

         PROCEDURE InsertSymbol (s:symbol) : HeadPtr;
         VAR p : HeadPtr;
         BEGIN
            p := clauses[s[1]];
            WHILE (p # NIL) AND (NOT EqStr(p^.sym,s)) DO p := p^.nxt; END;
            IF (p=NIL) THEN
               p := makenode(s);
               p^.nxt := clauses[s[1]];
               clauses[s[1]] := p;
            END;
            RETURN p;
         END InsertSymbol;


(* Output i space characters. For tabulation when listing the database *)

         PROCEDURE PrintSpcs (i: CARDINAL);
         VAR j : CARDINAL;
         BEGIN
            FOR j := 1 TO i DO WrChar(' '); END;
         END PrintSpcs;


(* List clauses *)

         PROCEDURE ListClauses (h : HeadPtr);
         VAR b : BodyPtr;
             a : AtomPtr;
             l : CARDINAL;
         BEGIN
         WHILE h # NIL DO
            b := h^.clause;
            WHILE b # NIL DO
               l := Length(h^.sym)+4;
               WrStr(h^.sym);
               a := b^.first;
               IF a # NIL THEN WrStr(' :- ') ELSE WrStr('.'); WrLn END;
               WHILE a # NIL DO
                  WrStr(a^.sym^.sym);
                  a := a^.nxt;
                  IF a # NIL THEN
                     WrStr(',');
                     WrLn;
                     PrintSpcs(l)
                  ELSE WrStr('.');
                       WrLn;
                  END;
               END;
```

```
            b := b^.nxt;
         END;
         h := h^.nxt;
      END;
      END ListClauses;


(* List all clauses in the database *)

      PROCEDURE listing;
      VAR  c : CHAR;
      BEGIN
         FOR c := 'a' TO 'z' DO
            ListClauses(clauses[c]);
         END;
      END listing;


(* --- module initialization ---------------------------------------------- *)

VAR c : CHAR;
BEGIN
   FOR c := 'a' TO 'z' DO clauses[c] := NIL;
   END;
   goal := makenode('[GOAL]');
END Dbase.
```

```
DEFINITION MODULE Lex;

IMPORT FIO;
FROM   Dbase IMPORT HeadPtr;

TYPE TknCls = (dot, com, col, eop, sym, err);

VAR Token : RECORD
               Class : TknCls;
               Inst  : HeadPtr;
            END;

    FPtr : FIO.File;

PROCEDURE GetToken;

END Lex.
```

```
IMPLEMENTATION MODULE Lex;


FROM Dbase IMPORT InsertSymbol, maxsymln, symbol;
IMPORT IO;

CONST   carret  = 15C;
        newln   = 12C;
        tab     = 11C;
        nullch  = 00C;

TYPE    charset = SET OF CHAR;

VAR unrd : BOOLEAN;
    c    : CHAR;


(* Get one character from the current input stream, or reread the
   last character read if the unrd flag is TRUE *)

        PROCEDURE getchar;
        BEGIN
           IF unrd THEN unrd := FALSE
           ELSE
              IF FPtr = FIO.StandardInput THEN c := IO.RdChar()
              ELSE
                 c := FIO.RdChar(FPtr);
                 IF FIO.EOF THEN c := nullch; END;
              END;
           END;
           IF c IN charset{carret,newln,tab} THEN c := ' ' END;
        END getchar;


(* Skip space characters in the input *)

        PROCEDURE skipspcs;
        BEGIN
              REPEAT getchar; UNTIL (c # ' ');
        END skipspcs;


(* Return next token in variable Token. *)

        PROCEDURE GetToken;
        VAR i : CARDINAL;
            s : symbol;
        BEGIN
           skipspcs;
           CASE c OF
              ',' : Token.Class := com;
           |  '.' : Token.Class := dot;
           |  nullch :
                    Token.Class := eop;
           |  ':' : getchar;
                    IF c='-' THEN
                       Token.Class := col;
                    ELSE
                       Token.Class := err
                    END;
           | 'a'..'z' :
                    i := 1;
                    WHILE (c IN charset{'a'..'z'}) AND (i<=maxsymln) DO
                       s[i] := c;
                       INC(i);
                       getchar;
                    END;
                    s[i] := 00C;
                    unrd := c#' ';
                    Token.Class := sym;
                    Token.Inst  := InsertSymbol(s);
```

```
            ELSE    Token.Class := err;
            END;
        END GetToken;

(* --- module initialization ---------------------------------------------- *)

BEGIN
   FPtr := FIO.StandardInput;
   unrd := FALSE;
END Lex.
```

```
MODULE PROPOS;

IMPORT FIO;
FROM IO     IMPORT WrStr, WrLn, WrChar, RdStr;
FROM AsmLib IMPORT ParamCount, ParamStr, DisableBreakCheck;
FROM Dbase  IMPORT symbol, HeadPtr, BodyPtr, AtomPtr, HeadRec, BodyRec,
                   AtomRec, NewHead, NewBody, NewAtom, listing,
                   ListClauses, DisposeClause, EqStr, goal;
FROM Lex    IMPORT TknCls, Token, GetToken, FPtr;

TYPE
    filename  =  ARRAY [1..40] OF CHAR;
    message   =  ARRAY [1..70] OF CHAR;

VAR FName : filename;
    Exit  : BOOLEAN;


(* Report error *)

        PROCEDURE error(m:message);
        BEGIN
           WrStr('*** ERROR: ');
           WrStr(m);
           WrLn;
        END error;


(* Attempt to prove the proposition pointed to by HeadPtr. *)

        PROCEDURE Prove (h:HeadPtr);

             PROCEDURE proveclause(h:HeadPtr) : BOOLEAN;
             VAR b : BodyPtr;
                 t : BOOLEAN;

                 PROCEDURE provebody(b:BodyPtr) : BOOLEAN;
                 VAR a : AtomPtr;
                     t : BOOLEAN;
                 BEGIN
                    t := TRUE;
                    a := b^.first;
                    WHILE (a # NIL) AND (t) DO
                       t := proveclause(a^.sym);
                       a := a^.nxt
                    END;
                    RETURN t;
                 END provebody;

             BEGIN
                t := FALSE;
                b := h^.clause;
                WHILE (b # NIL) AND ( NOT t) DO
                   t := provebody(b);
                   b := b^.nxt
                END;
                RETURN t;
             END proveclause;

        BEGIN
           ListClauses(goal);
           IF (h=NIL) OR (NOT proveclause(h)) THEN WrStr('NO')
                                              ELSE WrStr('YES'); END;
           WrLn;
        END Prove;


(* Process a command, introduced by a . *)

        PROCEDURE ProcessCommand () : BOOLEAN;
        VAR ok : BOOLEAN;
```

```
        BEGIN
           ok := TRUE;
           GetToken;
           IF    EqStr (Token.Inst^.sym,'listing') THEN
                 listing
           ELSIF EqStr (Token.Inst^.sym,'exit') THEN
                 Exit := TRUE;
           ELSIF EqStr (Token.Inst^.sym,'retract') THEN
                 GetToken;
                 DisposeClause (Token.Inst);
           ELSE  error('Unrecognized command');
                 ok := FALSE;
           END;
           RETURN ok;
        END ProcessCommand;


(* Construct a linked list of AtomRecs representing the body
   of a clause *)

        PROCEDURE FormBody(VAR a:AtomPtr) : BOOLEAN;
        VAR a1,a2 : AtomPtr;
        BEGIN
           a  := NIL;
           IF Token.Class=col THEN
           REPEAT
              GetToken;
              IF Token.Class=sym THEN
                 a2      := NewAtom();
                 a2^.nxt := NIL;
                 a2^.sym := Token.Inst;
                   IF a=NIL THEN a := a2
                   ELSE a1^.nxt := a2; END;
                 a1 := a2;
                 GetToken;
              END;
           UNTIL Token.Class # com;
           END;
           IF Token.Class # dot THEN error('. expected'); END;
           RETURN Token.Class=dot;
        END FormBody;


(* Construct a BodyRec for a new clause *)

        PROCEDURE ReadBody(p:HeadPtr) : BOOLEAN;
        VAR c,c1 : BodyPtr;
            a    : AtomPtr;
            b    : BOOLEAN;
        BEGIN
           c := p^.clause;
           c1 := NewBody();
           c1^.nxt := NIL;
           IF c = NIL THEN p^.clause := c1
           ELSE
              WHILE c^.nxt # NIL DO c:=c^.nxt END;
              c^.nxt := c1
           END;
           b := FormBody(a);
           c1^.first := a;
           RETURN b;
        END ReadBody;


(* Read in and process a clause, linking definite clauses into the
   database and processing queries and commands *)

        PROCEDURE ReadClause() : BOOLEAN;
        VAR h : HeadPtr;
        BEGIN
           GetToken;
```

```
            CASE Token.Class OF
              eop  :  RETURN FALSE;
            | dot  :  RETURN ProcessCommand ();
            | col  :  IF ReadBody(goal) THEN
                         Prove (goal);
                         DisposeClause(goal);
                         RETURN TRUE;
                      ELSE
                         error('Error in query');
                         RETURN FALSE;
                      END;
            | sym  :  h := Token.Inst;
                      GetToken;
                      RETURN ReadBody(h);
              ELSE
                error('Clause head expected');
                RETURN FALSE
              END;
          END ReadClause;


(* Read in a set of clauses from file *)

          PROCEDURE loadprog (fname : filename);
          VAR buffer : ARRAY [1..512+FIO.BufferOverhead] OF BYTE;
          BEGIN
             FPtr := FIO.Open(fname);
             FIO.AssignBuffer(FPtr,buffer);
             REPEAT UNTIL NOT ReadClause();
             FIO.Close(FPtr);
          END loadprog;


(* --- initialization and main loop ---------------------------------------- *)

BEGIN
   Exit := FALSE;
   DisableBreakCheck;
   IF ParamCount() > 0 THEN
      ParamStr(FName,1);
      loadprog(FName);
   END;
   FPtr := FIO.StandardInput;
   REPEAT
      WrLn;
      WrStr('> ');
      IF ReadClause() THEN ; END;
   UNTIL Exit;
END PROPOS.
```

APPENDIX B
VRP Source Code

VRP - MAIN MODULE

```
MODULE VRP;

(* VERY RUDIMENTARY PROLOG -- Startup module.
   Initializes debug flags, gets any command-line arguments, and
   enters the reader.
*)

IMPORT Parse;
FROM   DBase    IMPORT MEMUsage;
FROM   Streams  IMPORT ToTerm, FromTerm, ToPrinter, WriteLn, WrLn;
FROM   AsmLib   IMPORT ParamCount, ParamStr;
FROM   Inbuilt  IMPORT DefineInbuilts;
IMPORT STable;
IMPORT Lex;
IMPORT ProcGoal;
IMPORT Stack;


(* ------------------------------------------------------------------------ *)


VAR FName : ARRAY[1..100] OF CHAR;

BEGIN
   ToTerm;
   Lex.DBG        := FALSE;
   Parse.DBG      := FALSE;
   STable.DBG     := FALSE;
   ProcGoal.DBG   := FALSE;
   Stack.DBG      := FALSE;

   IF (ParamCount()=1) THEN
      ParamStr(FName,1)
   ELSE
      FName := "";
   END;
   WrLn;
   WriteLn('VRP - Very Rudimentary Prolog -- 1990');
   WrLn;
   DefineInbuilts;
   Parse.Reader (FName);
   ToTerm;
   FromTerm;
END VRP.
```

GLOBAL - DEFINITION

```
DEFINITION MODULE Global;

(* This module declares some global variables and types

   Exports:
        memory usage reporting
        Exit : is set to TRUE to flag an exit condition.
        Mode : is set to 'system' during the reading in of the predefined
               predicates. During user input, mode is set to 'user'.
               Predicates defined under 'system' may not be redefined in
               user mode.
*)


TYPE  mode = (system, user);

VAR   Exit : BOOLEAN;
      Mode : mode;


PROCEDURE MEMUsage;


END Global.
```

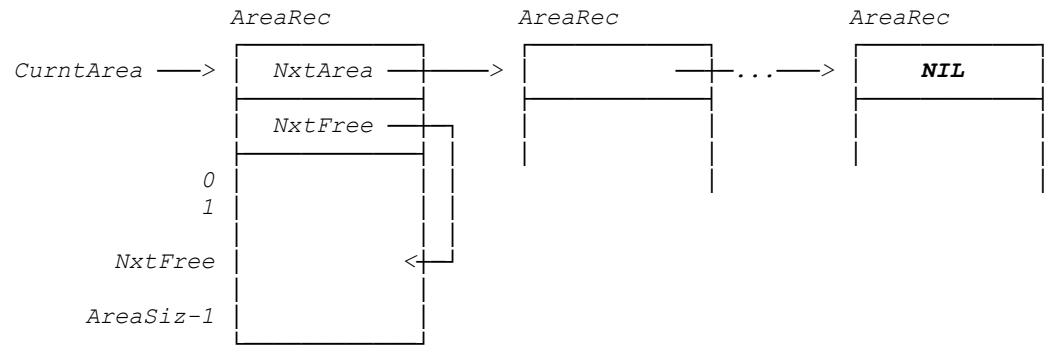
GLOBAL - IMPLEMENTATION

```
IMPLEMENTATION MODULE Global;

IMPORT DBase;
IMPORT Stack;
IMPORT Streams;
IMPORT Storage;


(* ------------------------------------------------------------------------ *)


PROCEDURE MEMUsage;
BEGIN
   DBase.MEMUsage;
   Stack.MEMUsage;
   Streams.WrStr     ("Largest block on heap : ");
   Streams.WrLngCard (LONGCARD(Storage.HeapAvail(Storage.MainHeap)) * 16,0);
   Streams.WriteLn   (' bytes.');
   Streams.WrStr     ("Total space on heap   : ");
   Streams.WrLngCard (LONGCARD(Storage.HeapTotalAvail(Storage.MainHeap)) * 16,0);
   Streams.WriteLn   (' bytes.');
END MEMUsage;


(* --- Module initialization ------------------------------------------- *)


BEGIN
   Exit := FALSE;
   Mode := system;
END Global.
```

SSTR - DEFINITION

```
DEFINITION MODULE Sstr;
(*-----------------------------------------------------------------------------
   String-storage manager
   ----------------------
   This module maintains a string-store. Strings passed to the module
   are stored in a string area, and a string pointer (Sptr) is returned
   to the caller.

   The Sptr is a pointer to an arry[0..MaxStrLen] of char. The string
   dereferenced by this pointer is NIL terminated so that it can be
   passed to procedures in the standard Str module, and to WrStr in IO
   module.

   It is up to the caller to impose a structure on the string buffer using
   the string pointers returned by procedure Sstore.

   The procedure Sclear deallocates the string store. The caller must
   ensure that no dangling pointers remain after a call to Sclear.
-----------------------------------------------------------------------------*)

CONST MaxStrLen = 250;
TYPE  Sptr      = POINTER TO ARRAY [0..MaxStrLen] OF CHAR;


(* Clear string store
   FUNCTION     Clears all strings in string-store and deallocates
                memory.
   CALL         Sclear();
*)
PROCEDURE Sclear();



(* Store String
   FUNCTION    Store string in string area, returning pointer to the
               string (Sptr), or NULL if insufficient space.
   CALL        Ptr := Sstore(S);
               Ptr is of type Sptr
               S is an array of char (variable only).
*)
PROCEDURE Sstore (VAR s : ARRAY OF CHAR) : Sptr;


END Sstr.
```

```
IMPLEMENTATION MODULE Sstr;

(*-----------------------------------------------------------------------------
 The string store is implemented as a linked list of string areas (AreaRec).
 Each area contains a 2-field header:

          NxtArea : pointer to next area
          NxtFree : index to the next free storage position in this area.

 The rest of the string area is an array of AreaSiz characters (Area).
 Strings are stored sequentially in the array, and terminated by a NULL
 character, as required by WrStr and the string processing procedures in
 module Str.

 Initially the string store consists of a single empty area. A new area is
 added to the HEAD of the list when the current one becomes full (ie when
 the length of the string to be stored exceeds the remaining space).

 Structure of String Store:
```




```
-----------------------------------------------------------------------------*)

FROM   Storage IMPORT ALLOCATE, DEALLOCATE;
FROM   SYSTEM  IMPORT TSIZE;
FROM   Str     IMPORT Length;

FROM   Streams IMPORT ReportErr;

(*-----------------------------------------------------------------------------*)

CONST AreaSiz = 1024;
(* Size for a string area. MUST EXCEED MaxStrLen BY AT LEAST 1 *)

TYPE AreaIndx = [0..AreaSiz-1];
TYPE AreaPtr = POINTER TO AreaRec;

TYPE AreaRec = RECORD
        NxtArea  : AreaPtr;
        NxtFree  : AreaIndx;
        Area     : ARRAY AreaIndx OF CHAR;
     END;

VAR CurntArea : AreaPtr;


(* ------------------------------------------------------------------------- *)


     (* Report error and halt *)
        PROCEDURE SstrErr();
        BEGIN
           ReportErr ("SSTR - Out of Memory");
           HALT;
```

```
      END SstrErr;


   (* Deallocate all string areas starting from the area pointed
      to by p. *)
      PROCEDURE DeallocArea (VAR p : AreaPtr);
      BEGIN
         IF (p <> NIL) THEN
            DeallocArea(p^.NxtArea);
            DEALLOCATE(p,TSIZE(AreaRec));
            p := NIL;
         END; (* IF *)
      END DeallocArea;


   (* Create a new string area and link into list *)
      PROCEDURE CreateNewArea() : AreaPtr;
      VAR p : AreaPtr;
      BEGIN
         ALLOCATE(p,TSIZE(AreaRec));
         IF (p <> NIL) THEN
            p^.NxtArea := NIL;
            p^.NxtFree  := 0;
         ELSE SstrErr
         END; (* IF *)
         RETURN p;
      END CreateNewArea;


   (* Clear all string areas forming the string store. One (empty)
      area is always retained. *)
      PROCEDURE Sclear;
      BEGIN
         CurntArea^.NxtFree := 0;
         DeallocArea(CurntArea^.NxtArea);
      END Sclear;


   (* Store a string. A pointer to the stored string is returned *)
      PROCEDURE Sstore(VAR s : ARRAY OF CHAR) : Sptr;
      VAR strlen : CARDINAL;
          INDX   : CARDINAL;
          t      : CARDINAL;
          p      : Sptr;
          Aptr   : AreaPtr;
      BEGIN
         strlen := Length(s);
         Aptr   := CurntArea;
         IF ((Aptr^.NxtFree + strlen) >= AreaSiz) THEN
            Aptr := CreateNewArea();
            Aptr^.NxtArea := CurntArea;
            CurntArea := Aptr;
         END; (* IF *)
         t := Aptr^.NxtFree;
         FOR INDX := 0 TO strlen-1 DO
             Aptr^.Area[t+INDX] := s[INDX];
         END;
         Aptr^.Area[t+strlen] := CHR(0);
         INC(Aptr^.NxtFree,strlen+1);
         RETURN Sptr(ADR(Aptr^.Area[t]));
      END Sstore;


(* --- Module initialization ---------------------------------------------- *)

BEGIN
   CurntArea := CreateNewArea();
END Sstr.
```

```
DEFINITION MODULE DBase;

(* Definition of database types and some operations.

   Exports:
        Symbol table, clause, and term types,
        their constructors and destructors.

        Various functions and predicates which operate on objects of the
        above type.

   NOTE that 'term' and 'term-record' refer to the basic internal structure
        used to construct the internal representation of clauses. These
        objects do not always coincide with terms as defined in the
        literature. Perhaps it would have been better had the non-committal
        terms 'object' and 'obj-record' been used.
*)


FROM Sstr    IMPORT Sptr;
FROM Global  IMPORT mode;
FROM Inbuilt IMPORT InBltProc;


CONST MaxVars = 1000;
      (* Max number of variables in a clause *)


TYPE SymType       = (variable, functor, anon, list);
     (* Symbol types : variable, functor, anonymous variable, and
        list constructor. *)


TYPE SymTabPtr     = POINTER TO SymTabRec;
     ClausePtr     = POINTER TO ClauseRec;
     TermPtr       = POINTER TO TermRec;


TYPE VarIndx       = CARDINAL [0..MaxVars];



(* --- Symbol-Table Record ------------------------------------------------
   One record for each variable and functor symbol in the database.
   In the case of functor symbols, the arity is also recorded, and
   pointers to the head and tail of the linked-list of clause with
   this functor as head are kept. The tail pointer is required for
   appending. The head pointer for traversing.

   A protection flag (mode) is used for protecting predefined system
   predicates from being redefined by the user or retracted. The
   count field has miscellaneous uses. In particular, it is used by
   the parses for mapping variables to stack-frame offsets.
*)

TYPE SymTabRec =
          RECORD
            Name    : Sptr;               (* Name pointer *)
            Next    : SymTabPtr;          (* Ptr to next entry *)
            Mode    : mode;               (* system or user defined *)
            Count   : CARDINAL;           (* For miscellaneous uses *)
            CASE SType : SymType OF
               functor  :
                  Arity   : SHORTCARD;    (* Arity of functor *)
                  FstCls  : ClausePtr;    (* Ptr to head of clause list *)
                  LstCls  : ClausePtr;    (* Ptr to tail of clause list *)
             |  variable :
```

```
          END
        END;


(* --- Term Record ------------------------------------------------------
   One record for each term in a clause.
   It is necessary to include the symbol type within TermRec because, although
   in the case of variables and functors the type can be read from the
   symbol-table entry, in the case of special terms such as list constructors
   and anonymous variables a symbol-table entry does not exist. Also, including
   the symbol type here makes for less dereferncing when examining terms.
*)

TYPE TermRec  =
        RECORD
          Next   : TermPtr;                (* Ptr to next TermRec in a terms list
*)
          Entry : SymTabPtr;              (* Ptr to ST entry for this term *)
          CASE  SType  : SymType OF
             list,functor : Args : TermPtr;  (* Ptr to argument list *)
          |  variable     : Ofst : VarIndx;  (* Offset of var within stack frame
*)
          END
        END;


(* --- Clause Record ----------------------------------------------------
   One for every clause in the database. The Head field points to the term
   record at the head of the clause, while the Body field points to a linked
   list of the terms in the body of the clause. The Vars field records the
   number of variables in the clause, required to calculate the stack frame
   size.
*)

TYPE ClauseRec =
        RECORD
          Next  : ClausePtr;              (* Ptr to next clause *)
          CASE InBlt : BOOLEAN OF
            TRUE  : Proc  : InBltProc;    (* An inbuilt procedure *)
                    Entry : SymTabPtr;
          | FALSE : Vars  : VarIndx;      (* Number of variables in clause *)
                    Head  : TermPtr;      (* Ptr to term at head of clause *)
                    Body  : TermPtr;      (* Ptr to term list forming body *)
          END;
        END;


(* --- Predicates & functions -------------------------------------------

   The following predicates and functions are defined on the above data
   structures.

*)


PROCEDURE IsVar   (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term is a variable symbol *)

PROCEDURE IsFunctor  (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term is a functor, ie predicate, function or
   constant symbol *)

PROCEDURE IsConst (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term is a functor of arity 0 with which
   no clauses are associated - ie does not appear as the head of any clause *)

PROCEDURE IsPred (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term is a functor which appears as the head of some clause *)
```

```
PROCEDURE IsAnon (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term is the anonymous variable *)

PROCEDURE IsList (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term represents a list *)

PROCEDURE IsNulList (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term represents a Null list *)

PROCEDURE IsNonNulList (TPtr : TermPtr) : BOOLEAN;
(* TRUE if term is a list but is not the Null list *)



PROCEDURE IsAssertion (CPtr : ClausePtr) : BOOLEAN;
(* TRUE if clause is an assertion, ie has a null body *)

PROCEDURE GetAry  (TPtr : TermPtr) : SHORTCARD;
(* Returns arity of term if a functor. Otherwise not defined *)

PROCEDURE GetFunctor (TPtr : TermPtr) : SymTabPtr;
(* Returns pointer to symbol table entry of principle functor of a
   TermRec. Returns NIL if TermRec is not a functor. *)

PROCEDURE SameFunctor (L1,L2 : TermPtr) : BOOLEAN;
(* TRUE if L1,L2 are
        both empty lists, OR
        both non-empty lists, OR
        both structures with the same functor.
*)



(* *** OBJECT CONSTRUCTORS and DESTRUCTORS ******************************** *)

(* --- Constructors ------------------------------------------------------
   Constructors return NIL if the construction of an object fails.
*)

PROCEDURE MKSymTabRec () : SymTabPtr;
PROCEDURE MKClauseRec () : ClausePtr;
PROCEDURE MKTermRec   (SType : SymType) : TermPtr;



(* --- Destructors -------------------------------------------------------
   Destructors deallocate the memory used by an object and all its
   subobjects, as follows:

   RMTermRec    :  Removes a TermRec and all its argument records.
   RMTermList   :  Removes a linked list of TermRec and their argument records.
   RMClauseRec  :  Removes a ClauseRec, its head predicate, and all the
                   terms (literals) in its body.
   RMSymTabRec  :  Removes a SymTabRec and all its clauses.

*)

PROCEDURE RMTermRec   (TPtr : TermPtr);
PROCEDURE RMTermList  (TPtr : TermPtr);
PROCEDURE RMClauseRec (CPtr : ClausePtr);
PROCEDURE RMSymTabRec (SPtr : SymTabPtr);


PROCEDURE MEMUsage;

END DBase.
```

```
IMPLEMENTATION MODULE DBase;

FROM Storage IMPORT ALLOCATE, Available;
FROM SYSTEM  IMPORT TSIZE;
FROM AsmLib  IMPORT AddAddr;
FROM Streams IMPORT WrStr, WrCard, WrLn, ReportErr;

TYPE MemUsage = RECORD
                  FreeList : ADDRESS;
                  Used     : CARDINAL;
                  Free     : CARDINAL;
                END;



CONST SZSymTabRec = TSIZE (SymTabRec);
      SZClauseRec = TSIZE (ClauseRec);
      SZTermRec    = TSIZE (TermRec);

VAR   MEMSymTab    : MemUsage;
      MEMClause    : MemUsage;
      MEMTerm       : MemUsage;


(* ------------------------------------------------------------------------ *)

(* Term predicates - test class of a TermRec, returning BOOLEAN.
      IsVar          - variable object
      IsFunctor      - functor object (could be predicate, function or constant
      IsConst        - functor of arity 0 and not head of a clause list
      IsPred         - functor, head of a clause list
      IsAnon         - object representing the anonymous variable
      IsList         - object representing the list constructor
      IsNulList      - object representing the null list
      IsNonNulList   - list object, but not representing the null list
*)

        PROCEDURE IsVar   (TPtr : TermPtr) : BOOLEAN;
        BEGIN
           RETURN TPtr^.SType = variable;
        END IsVar;


        PROCEDURE IsFunctor  (TPtr : TermPtr) : BOOLEAN;
        BEGIN
           RETURN TPtr^.SType = functor;
        END IsFunctor;


        PROCEDURE IsConst (TPtr : TermPtr) : BOOLEAN;
        BEGIN
           RETURN (TPtr^.SType = functor)    AND
                  (TPtr^.Entry^.Arity = 0)   AND
                  (TPtr^.Entry^.FstCls = NIL);
        END IsConst;


        PROCEDURE IsPred (TPtr : TermPtr) : BOOLEAN;
        BEGIN
           RETURN (TPtr^.SType = functor)      AND
                  (TPtr^.Entry^.FstCls # NIL);
        END IsPred;


        PROCEDURE IsAnon (TPtr : TermPtr) : BOOLEAN;
        BEGIN
           RETURN (TPtr^.SType = anon);
```

```
        END IsAnon;


        PROCEDURE IsList (TPtr : TermPtr) : BOOLEAN;
        BEGIN
            RETURN TPtr^.SType = list;
        END IsList;


        PROCEDURE IsNulList (TPtr : TermPtr) : BOOLEAN;
        BEGIN
           RETURN IsList(TPtr) AND (TPtr^.Args = NIL);
        END IsNulList;


        PROCEDURE IsNonNulList (TPtr : TermPtr) : BOOLEAN;
        BEGIN
           RETURN IsList(TPtr) AND (TPtr^.Args # NIL);
        END IsNonNulList;



(* ------------ Miscellaneous predicates and functions -------------------- *)

(* Tests whether a definite clause is a unit clause.
   TRUE if clause has a NULL body
*)

        PROCEDURE IsAssertion (CPtr : ClausePtr) : BOOLEAN;
        BEGIN
           RETURN CPtr^.Body = NIL;
        END IsAssertion;



(* Returns the arity of a term if the term represents a functor.
   Otherwise undefined.
*)

        PROCEDURE GetAry  (TPtr : TermPtr) : SHORTCARD;
        BEGIN
           RETURN TPtr^.Entry^.Arity;
        END GetAry;



(* Returns pointer to symbol table entry of principle functor of a
   TermRec. Returns NIL if TermRec is not a functor. *)

        PROCEDURE GetFunctor (TPtr : TermPtr) : SymTabPtr;
        BEGIN
           IF IsFunctor(TPtr) THEN
              RETURN TPtr^.Entry;
           ELSE RETURN NIL;
           END;
        END GetFunctor;



(* TRUE if L1,L2 are
        both empty lists, OR
        both non-empty lists, OR
        both structures with the same functor.
*)

        PROCEDURE SameFunctor (L1,L2 : TermPtr) : BOOLEAN;
        BEGIN
           RETURN
              (IsNulList(L1) AND IsNulList(L2)) OR
              (IsNonNulList(L1) AND IsNonNulList(L2)) OR
              (   (L1^.SType = functor) AND (L2^.SType = functor) AND
```

```
                (L1^.Entry = L2^.Entry)
            )
        END SameFunctor;


(* ------------- INTERNAL MEMORY ALLOCATION PRIMITIVES -------------------- *)

(* The module maintains a free list of each of the tree types of structures
   required by the database.  When a freelist becomes exhausted, a chunk of
   memory is requested from the system and divided into structure-sized
   units which are linked to form a new free list. This is done because:
   1.  system storage requests carry a not-insignificant time overhead.
       This is evident from examination of the code for the Storage module.
   2.  the Storage module can only allocate memory in paragraph-sized chunks
       (16-bytes). Had structures to be allocated individually from the system
       heap, the wastage per unit allocation would be as follows:

       Structure    size (bytes)   allocation (paras)   wastage (bytes)
       SymTabRec    20             2                    12
       TermRec      13             1                    3
       ClauseRec    15             1                    1
*)

        PROCEDURE AllocSymTab (Qty : CARDINAL) : BOOLEAN;
        VAR SPtr  : SymTabPtr;
            I     : CARDINAL;
        BEGIN
           I := SZSymTabRec * Qty;
           IF NOT Available(I) THEN
              RETURN FALSE;
           END;
           ALLOCATE (SPtr, I);
           MEMSymTab.FreeList := SPtr;
           MEMSymTab.Free     := Qty;
           FOR I := 1 TO Qty-1 DO
               SPtr^.Next := AddAddr(SPtr,SZSymTabRec);
               SPtr := SPtr^.Next;
           END; (*FOR*)
           SPtr^.Next := NIL;
           RETURN TRUE;
        END AllocSymTab;



        PROCEDURE AllocClause (Qty : CARDINAL) : BOOLEAN;
        VAR CPtr  : ClausePtr;
            I     : CARDINAL;
        BEGIN
           I := SZClauseRec * Qty;
           IF NOT Available(I) THEN
              RETURN FALSE;
           END;
           ALLOCATE (CPtr, I);
           MEMClause.FreeList := CPtr;
           MEMClause.Free     := Qty;
           FOR I := 1 TO Qty-1 DO
               CPtr^.Next := AddAddr(CPtr,SZClauseRec);
               CPtr := CPtr^.Next;
           END; (*FOR*)
           CPtr^.Next := NIL;
           RETURN TRUE;
        END AllocClause;



        PROCEDURE AllocTerm (Qty : CARDINAL) : BOOLEAN;
        VAR TPtr  : TermPtr;
            I     : CARDINAL;
        BEGIN
           I := SZTermRec * Qty;
           IF NOT Available(I) THEN
```

```
            RETURN FALSE;
         END;
         ALLOCATE (TPtr, I);
         MEMTerm.FreeList := TPtr;
         MEMTerm.Free     := Qty;
         FOR I := 1 TO Qty-1 DO
             TPtr^.Next := AddAddr(TPtr,SZTermRec);
             TPtr := TPtr^.Next;
         END; (*FOR*)
         TPtr^.Next := NIL;
         RETURN TRUE;
      END AllocTerm;


(* --------- STRUCTURE CONSTRUCTORS and DESTRUCTORS ----------------------- *)

(*
   Constructors return NIL if the construction of a structure fails.

   Destructors deallocate the memory used by an object and all its
   subobjects, as follows:

   RMTermRec    :  Removes a TermRec and all its argument records.
   RMTermList   :  Removes a linked list of TermRec and their argument records.
   RMClauseRec  :  Removes a ClauseRec, its head predicate, and all the
                   terms (literals) in its body.
   RMSymTabRec  :  Removes a SymTabRec and all its clauses.

*)

      PROCEDURE MKSymTabRec () : SymTabPtr;
      VAR SPtr : SymTabPtr;
      BEGIN
         IF MEMSymTab.Free=0 THEN
            IF NOT AllocSymTab (20)
            THEN RETURN NIL
            END
         END;
         SPtr := MEMSymTab.FreeList;
         MEMSymTab.FreeList := SPtr^.Next;
         INC (MEMSymTab.Used);
         DEC (MEMSymTab.Free);
         RETURN SPtr;
      END MKSymTabRec;


      PROCEDURE MKClauseRec () : ClausePtr;
      VAR CPtr : ClausePtr;
      BEGIN
         IF MEMClause.Free=0 THEN
            IF NOT AllocClause (10)
            THEN RETURN NIL
            END
         END;
         CPtr := MEMClause.FreeList;
         MEMClause.FreeList := CPtr^.Next;
         INC (MEMClause.Used);
         DEC (MEMClause.Free);
         RETURN CPtr;
      END MKClauseRec;


      PROCEDURE MKTermRec (SType : SymType) : TermPtr;
      VAR TPtr : TermPtr;
      BEGIN
         IF MEMTerm.Free=0 THEN
            IF NOT AllocTerm (50)
            THEN RETURN NIL
            END
         END;
```

```
      TPtr := MEMTerm.FreeList;
      MEMTerm.FreeList := TPtr^.Next;
      TPtr^.SType := SType;
      INC (MEMTerm.Used);
      DEC (MEMTerm.Free);
      RETURN TPtr;
   END MKTermRec;


   PROCEDURE RMTermList (TPtr : TermPtr);
   VAR TPtr2 : TermPtr;
   BEGIN
      WHILE TPtr # NIL DO
         TPtr2 := TPtr^.Next;
         RMTermRec (TPtr);
         TPtr := TPtr2;
      END;
   END RMTermList;


   PROCEDURE RMTermRec (TPtr : TermPtr);
   BEGIN
      IF (TPtr^.SType = functor) OR (TPtr^.SType = list) THEN
         RMTermList (TPtr^.Args);
      END;
      TPtr^.Next := MEMTerm.FreeList;
      MEMTerm.FreeList := TPtr;
      INC (MEMTerm.Free);
      DEC (MEMTerm.Used);
   END RMTermRec;


   PROCEDURE RMClauseRec (CPtr : ClausePtr);
   VAR TPtr : TermPtr;
   BEGIN
      RMTermRec(CPtr^.Head);
      RMTermList(CPtr^.Body);
      TPtr^.Next := MEMClause.FreeList;
      MEMClause.FreeList := CPtr;
      INC(MEMClause.Free);
      DEC(MEMClause.Used);
   END RMClauseRec;


(* Since no provision currently exists in the interpreter to clear the
   database, this procedure has not yet been implemented.

   Note that, while deallocating a single symbol-table entry is easy,
   ensuring that the database contains no references to the entry is
   far from straightforward. It may be prefereable to retain a few
   redundant entries than implement the check for dangling pointers.
*)

   PROCEDURE RMSymTabRec (SPtr : SymTabPtr);
   BEGIN
   END RMSymTabRec;



(* ------------------------------------------------------------------------ *)

(* Report on database memory usage *)

   PROCEDURE MEMUsage;
   BEGIN
      WrStr ('SymTab : Used=') ; WrCard (MEMSymTab.Used,0);
      WrStr ('  Free=')        ; WrCard (MEMSymTab.Free,0)  ; WrLn ;
      WrStr ('Clause : Used=') ; WrCard (MEMClause.Used,0);
      WrStr ('  Free=')        ; WrCard (MEMClause.Free,0)  ; WrLn ;
      WrStr ('Term   : Used=') ; WrCard (MEMTerm.Used,0);
      WrStr ('  Free=')        ; WrCard (MEMTerm.Free,0)    ; WrLn ;
```

```
      END MEMUsage;


(* --- module initialization ---------------------------------------------- *)

(* Allocates free lists for the database structures.
   If insufficient memory, then program halts, since it is pointless
   to continue if no database can be allocated.
*)

BEGIN
   IF AllocSymTab (100) AND
      AllocClause (50)  AND
      AllocTerm   (200)
   THEN
      MEMSymTab.Used := 0;
      MEMClause.Used := 0;
      MEMTerm.Used   := 0;
   ELSE
      ReportErr ('Insufficient memory to run');
      HALT;
   END;
END DBase.
```

STABLE - DEFINITION

```
DEFINITION MODULE STable;

(* Symbol table module.

   Exports:
        Symbol table insertion routine
        lexicographic order between symbol table entries
        Database listing utilities.
*)


FROM DBase IMPORT SymType, SymTabPtr, TermPtr, ClausePtr;


(* ------------------------------------------------------------------------ *)

CONST NoCount = MAX(SHORTCARD);          (* Default value of count field *)

TYPE  order   = (lt,gt,le,ge);

VAR   DBG     : BOOLEAN;



(* Insert a new Symbol-Table Record for the object described by the parameters
   parameters if one does not already exist. (if SType is 'variable', arity is
   ignored).

   A pointer to the SymTabRec for the object is returned.
*)

PROCEDURE Insert    ( Name  : ARRAY OF CHAR;
                      SType : SymType;
                      Arity : SHORTCARD) : SymTabPtr;



(* Tests whether the two symbol-table entries pointed to by SP1 and SP1
   are in the lexicographic order SP1 Ord SP2
*)

PROCEDURE Test (SP1,SP2 : SymTabPtr;
                Ord     : order) : BOOLEAN;



(* Symbol table dump utility *)

PROCEDURE DumpST;


(* Term, clause, and database listing utilities *)

PROCEDURE ListTerm (TPtr : TermPtr);
PROCEDURE ListClause (CPtr : ClausePtr);
PROCEDURE ListDBase;


END STable.
```

STABLE - IMPLEMENTATION

```
IMPLEMENTATION MODULE STable;

FROM Sstr    IMPORT Sptr, Sstore;
FROM DBase   IMPORT SymTabRec, IsFunctor, IsConst, IsList, IsNonNulList,
                    IsNulList, MKSymTabRec;
FROM SYSTEM  IMPORT TSIZE;
FROM Str     IMPORT Compare;
FROM Streams IMPORT WrStr, WrCharRep, WrCard, WrShtCard, WrLn, WriteLn;
FROM Global  IMPORT Mode, mode;


(* ------------------------------------------------------------------------ *)

CONST LoChar = ' ';
      HiChar = '~';

VAR SymTab : ARRAY [LoChar..HiChar] OF SymTabPtr;


(* ------------------------------------------------------------------------ *)



(* --- InitST ---------------------------------------------------------------
   This procedure resets all the base pointers in the SymTab array
   to NUL. It does not deallocate the memory occupied by any currently
   resident database.
*)

        PROCEDURE InitST;
        VAR C : CHAR;
        BEGIN
           FOR C := LoChar TO HiChar DO
              SymTab [C] := NIL;
           END;
        END InitST;



(* --- Search ---------------------------------------------------------------
   Searches for a ST entry of the required name, type and (if applicable)
   arity.
   RETURNS:
      TRUE  if a match was found. Ptr is set to point to the matching entry.
      FALSE if no match found. In this case, Ptr points to the entry at which
            the search failed, ie the entry which should have preceeded the
            entry required. If the new record is to be inserted, then it
            should follow the record pointed to by Ptr. If Ptr is NIL, then
            the record should be inserted at the head of the list. If the
            Name is already in the string store, then NmPtr is set to
            point to the stored string. Otherwise NmPtr is set to NIL. In
            this case, the insertion routine should call Sstore to insert
            the new name in the string store.
*)

        PROCEDURE  Search   (    Name  : ARRAY OF CHAR;
                                 SType : SymType;
                                 Arity : SHORTCARD;
                            VAR NmPtr : Sptr;
                            VAR Ptr   : SymTabPtr) : BOOLEAN;

        CONST Equal   =  0;
              FstSmlr = -1;
              FstGrtr =  1;

        VAR SPtr   : SymTabPtr;
```

```
        BEGIN
           Ptr   := NIL;
           SPtr  := SymTab[Name[0]];
           NmPtr := NIL;

           LOOP
             IF SPtr = NIL THEN RETURN FALSE; END;
             CASE Compare(Name,SPtr^.Name^) OF
               Equal   :  NmPtr := SPtr^.Name;
                          IF (SType=variable) OR
                             ((SType=functor) AND (Arity=SPtr^.Arity)) THEN
                             Ptr := SPtr;
                             RETURN TRUE;
                          ELSIF (Arity < SPtr^.Arity) THEN
                             RETURN FALSE;
                          END;
             | FstSmlr :  RETURN FALSE;
             END; (*CASE*)
             Ptr  := SPtr;
             SPtr := SPtr^.Next;
           END; (*LOOP*)

        END Search;



(* --- Insert ------------------------------------------------------------
   If not already in symbol table, insert a new Symbol-Table Record with the
   details passed as parameters (if SType is 'variable', arity is ignored).

   RETURNS  Pointer to symbol table entry.

   The routine uses Search to determine whether a new record needs to be
   added to the symbol table. If a new record needs to be created, Search
   also indicates where the record has to be inserted to keep the table
   sorted, as well as whether the name string needs to be stored in the
   string store.

   The new record has its Mode field set to the current mode (system or
   user). The count field is initialized to NoCount.
*)

        PROCEDURE Insert    (Name  : ARRAY OF CHAR;
                             SType : SymType;
                             Arity : SHORTCARD) : SymTabPtr;

        VAR Ptr,P : SymTabPtr;
            NmPtr : Sptr;

        BEGIN
            IF Search(Name,SType,Arity,NmPtr,Ptr) THEN
               RETURN Ptr;
            END;

            (* If Search found no record with the same name, then the name must be
               saved in the string store *)
            IF NmPtr = NIL THEN
               NmPtr := Sstore(Name);
            END;

            (* Create a new record and initialize it *)
            P := MKSymTabRec ();
            P^.Name  := NmPtr;
            P^.SType := SType;
            P^.Mode  := Mode;
            P^.Count := NoCount;

            IF (SType=functor) THEN
               P^.Arity  := Arity;
               P^.FstCls := NIL;
```

```
            P^.LstCls := NIL;
         END;

         (* Link in the new record at the position indicated by Search in
            the variable Ptr.
            If Ptr is NIL, then the record is to be inserted at the head
            of the list for this hash group. Otherwise, the record is to
            be inserted following the record pointed to by Ptr.
         *)
         IF Ptr = NIL THEN
            P^.Next := SymTab[Name[0]];
            SymTab[Name[0]] := P;
         ELSE
            P^.Next := Ptr^.Next;
            Ptr^.Next := P;
         END;

       RETURN P;
      END Insert;


(* --- Test ------------------------------------------------------------
   Tests whether the two symbol-table entries pointed to by SP1 and SP2
   are in the lexicographic order SP1 Ord SP2
*)

      PROCEDURE Test (SP1,SP2 : SymTabPtr;
                      Ord     : order) : BOOLEAN;
      BEGIN
         CASE Ord OF
            lt : RETURN Compare(SP1^.Name^,SP2^.Name^) = -1;
         |  gt : RETURN Compare(SP1^.Name^,SP2^.Name^) =  1;
         |  le : RETURN Compare(SP1^.Name^,SP2^.Name^) <  1;
         |  ge : RETURN Compare(SP1^.Name^,SP2^.Name^) > -1;
         END;
      END Test;



(* --- DumpST ----------------------------------------------------------
   Dumps contents of symbol table to screen.
*)


      PROCEDURE DumpST;
      VAR C : CHAR;
          P : SymTabPtr;
      BEGIN
         FOR C := LoChar TO HiChar DO
         P := SymTab[C];
         WHILE P # NIL DO
           WrStr( P^.Name^);
           IF P^.SType = functor THEN
              WrStr( '(fnctr)');
              WrShtCard( P^.Arity,4);
           ELSE WrStr( '(vrbl)');
           END;
           WrLn ;
           P := P^.Next;
         END;
         END;
      END DumpST;



(* --- DATABASE LISTING ROUTINES ---------------------------------------

   Comprising:
        ListTerm   -  recursively lists a term and its arguments
        ListClause -  lists a clause (head and body).
        ListDBase  -  lists the database of clauses.
```

```
*)

        PROCEDURE ListTerm (TPtr : TermPtr);

             PROCEDURE ListArgs (TPtr : TermPtr);
             BEGIN
                TPtr := TPtr^.Args;
                IF TPtr=NIL THEN RETURN END;
                WrStr('(');
                WHILE TPtr#NIL DO
                   ListTerm(TPtr);
                   TPtr := TPtr^.Next;
                   IF TPtr#NIL THEN WrStr(',') END;
                END;
                WrStr(')');
             END ListArgs;

        BEGIN
           CASE TPtr^.SType OF
              variable  :  WrStr(TPtr^.Entry^.Name^);
                           IF DBG THEN
                              WrStr('(_');
                              WrCard(TPtr^.Ofst,1);
                              WrStr(')');
                           END;
           |  anon      :  WrStr('_');
           |  functor   :  WrStr(TPtr^.Entry^.Name^);
                           ListArgs (TPtr);
           |  list      :  WrStr('[');
                           WHILE IsNonNulList(TPtr) DO
                              TPtr := TPtr^.Args;
                              ListTerm (TPtr);
                              TPtr := TPtr^.Next;
                              IF IsNonNulList (TPtr) THEN
                                 WrStr (',');
                              END; (*IF*)
                           END; (*WHILE*)
                           IF IsNulList(TPtr) THEN
                              WrStr (']')
                           ELSE
                              WrStr('|');
                              ListTerm(TPtr);
                              WrStr(']');
                           END; (*IF*)
           END; (*CASE*)
        END ListTerm;

(* ------------------------------------------------------------------------ *)


        PROCEDURE ListClause (CPtr : ClausePtr);
        VAR TPtr : TermPtr;
        BEGIN

           IF CPtr^.InBlt THEN
              WrStr('* ');
              WrStr(CPtr^.Entry^.Name^);
              WrStr('/');
              WrShtCard (CPtr^.Entry^.Arity,0);
              WrLn;
              RETURN;
           END;

           IF DBG THEN
              WrStr('Vars = '); WrCard(CPtr^.Vars,2);WrLn;
           END;
           IF CPtr^.Head = NIL THEN
              WrStr('GOAL ')
           ELSE
              IF CPtr^.Head^.Entry^.Mode = system THEN WrStr('# '); END;
              ListTerm(CPtr^.Head);
```

```
            END; (*IF*)
            TPtr := CPtr^.Body;
            IF TPtr # NIL THEN
               WrStr(' :- ');
               REPEAT
                  ListTerm(TPtr);
                  TPtr := TPtr^.Next;
                  IF TPtr # NIL THEN
                     WriteLn(' ,'); WrCharRep(' ',10);
                  END;
               UNTIL TPtr = NIL;
            END;
            WriteLn(' .');
         END ListClause;

(* ------------------------------------------------------------------------ *)


         PROCEDURE ListDBase;
         VAR C : CHAR;
             CPtr : ClausePtr;
             SPtr : SymTabPtr;

         BEGIN
            FOR C := LoChar TO HiChar DO
               SPtr := SymTab[C];
               WHILE SPtr # NIL DO
                  IF (SPtr^.SType = functor) AND (SPtr^.FstCls # NIL) THEN
                  CPtr := SPtr^.FstCls;
                  REPEAT
                     ListClause (CPtr);
                     CPtr := CPtr^.Next;
                  UNTIL CPtr = NIL;
                  WrLn ;
                  END;
                  SPtr := SPtr^.Next;
               END;
            END;
         END ListDBase;


(* --- Module initialization ------------------------------------------- *)


BEGIN
   InitST;
   DBG     := FALSE;
END STable.
```

STACK - DEFINITION

```
DEFINITION MODULE Stack;

(* Runtime stack and trail manager.

   Exports:
        Frame type.
        Procedures to open stack, push and pop frames, record
        variable bindings on trail, reset bindings during
        backtracking, dereferencing bindings.
*)


FROM DBase IMPORT ClausePtr, TermPtr, MaxVars, VarIndx;


(* ------------------------------------------------------------------------ *)


CONST DefaultStkSz = 100000;

TYPE FramePtr  = POINTER TO Frame;
     BindPtr   = POINTER TO Binding;
     TrailPtr  = POINTER TO BindPtr;
     BindType  = (free,var,lit);


     Binding  = RECORD
                  CASE BType   : BindType OF
                       var     : BPtr : BindPtr;
                  |    lit     : TPtr : TermPtr;
                                 Env  : FramePtr;
                  END;
                END;

     Frame    = RECORD
                  Num        : CARDINAL;     (* Frame number. For debugging *)
                  CrntCls    : ClausePtr;    (* Current clause. "    "      *)

                  Prev       : FramePtr;
                  Vars       : VarIndx;
                  Parent     : FramePtr;
                  CrntLit    : TermPtr;
                  NxtClause  : ClausePtr;
                  CrntBTP    : FramePtr;
                  Trail      : TrailPtr;
                  Binds      : ARRAY [0..MaxVars-1] OF Binding;
                END;


VAR DBG       : BOOLEAN;

        PROCEDURE MAKEFrame    (Vars : VarIndx) : FramePtr;
        PROCEDURE PUSHFrame ;
        PROCEDURE POPFrames    (FPtr : FramePtr);
        PROCEDURE STORETrail   (BPtr : BindPtr) : BOOLEAN;
        PROCEDURE RESTORETrail (TPtr : TrailPtr);

        PROCEDURE GetStkTop () : FramePtr;

        PROCEDURE BindAdr  (FPtr : FramePtr; TPtr : TermPtr) : BindPtr;
        PROCEDURE IsFree   (BPtr : BindPtr)  : BOOLEAN;
        PROCEDURE DeRef    (BPtr : BindPtr)  : BindPtr;
        PROCEDURE NextCall (FPtr : FramePtr) : TermPtr;


        (* Given a TermPtr, return the TermPtr to which it is bound. A
           non-variable term is bound to itself. A variable-term is
```

```
            dereferenced, and the term to which it is bound is returned,
            or a NIL if the variable is free
         *)
         PROCEDURE GetBoundTerm (TPtr : TermPtr; FPtr : FramePtr) : TermPtr;


         PROCEDURE SetStackSize (InBytes : LONGCARD);
         PROCEDURE OPENStack;
         PROCEDURE CLOSEStack;
         (* Free all records on the stack and on the trail *)

         PROCEDURE Before(A1,A2 : ADDRESS) : BOOLEAN;
         (* Returns true if stack address A1 preceeds stack address A2 *)


         PROCEDURE MEMUsage;
         (* Report on memory usage *)

END Stack.
```

STACK - IMPLEMENTATION

```
IMPLEMENTATION MODULE Stack;

(* Stack manager - manages the procedure activation and trail stacks.
   These two stacks share a common block of memory and grow towards
   each other from opposite ends of the block.

   While trail records are of uniform size, the procedure activation
   frames held on the runtime stack are variable-sized. Because of this,
   and because a temporary frame is needed by the unification process,
   the two stacks are handled very differently. In particular, trailtop
   points to the NEXT free location on the trail, while stacktop points
   to the LAST OCCUPIED location on the stack. Also, frame records
   require a pointer to the previous record on the stack.

   The main pointers associated with the two stacks are:

                        +--------+ <-- TrailBase
                        +--------+
                        +--------+ <-- TrailTop
                        |        |
                        |        |
                        +........+ <-- StackEnd
                        +--------+ <-- NewFrame
                        +--------+ <-- StackTop
                        +--------+
                        +--------+ <-- StackBase

   NewFrame is required by the unification procedure. At the start of
   a unification, a NewFrame of the required size is allocated just
   beneath StackTop. If the unification succeeds, NewFrame is pushed
   onto the stack and becomes the new StackTop. Otherwise it is
   overwritten by the subsequent creation of a new NewFrame. StackEnd
   keeps track of the location where NewFrame ends, and is required
   in checking for collisions between the procedure-activation stack
   and the trail.

*)


FROM Storage IMPORT MainHeap, HeapAllocate, HeapDeallocate, HeapAvail;
FROM SYSTEM  IMPORT TSIZE,Seg,Ofs;
FROM AsmLib  IMPORT AddAddr,DecAddr;
FROM Streams IMPORT WrStr, WrCard, WrLn, WriteLn, WrLngCard, WrAddr;
FROM DBase   IMPORT IsVar;

IMPORT IO;


(* ------------------------------------------------------------------------ *)

CONST SZBind   = TSIZE(Binding);
      (* Size of a variable binding record *)

      SZFullFrame = TSIZE(Frame);
      (* Size of a frame with MaxVars variable bindings *)

      SZFrame0 = SZFullFrame - (SZBind * MaxVars);
      (* Size of a frame record with 0 variable bindings *)

      SZTrail  = TSIZE(BindPtr);
      (* Size of a trail record *)


VAR   SZStack   : CARDINAL; (* Stack size in paragraphs *)
      StackBase : FramePtr; (* Base address of stack *)
      TrailBase : TrailPtr; (* Base address of trail *)
      TrailTop  : TrailPtr; (* Address of next free location on trail *)
```

```
      StackTop  : FramePtr; (* Address of frame on top of stack *)
      NewFrame  : FramePtr; (* Frame following stacktop *)
      StackEnd  : ADDRESS;  (* Address following NewFrame *)

      StackOpen : BOOLEAN;  (* TRUE if stack currently allocated *)


(* ------------------------------------------------------------------------ *)


(* Return the number of bytes between two addresses, A2 >= A1 *)

        PROCEDURE DiffAddr (A1,A2 : ADDRESS) : LONGCARD;
        BEGIN
            RETURN LONGCARD(Seg(A2^) - Seg(A1^) ) * 16  +
                   LONGCARD(Ofs(A2^)) - LONGCARD(Ofs(A1^)) + 1;
        END DiffAddr;


(* Returns true if stack address A1 preceeds stack address A2 *)

        PROCEDURE Before (A1,A2 : ADDRESS) : BOOLEAN;
        BEGIN
           RETURN ( Seg(A1^) < Seg(A2^) ) OR
                  ( ( Seg(A1^) = Seg(A2^) ) AND
                    ( Ofs(A1^) < Ofs(A2^) )
                  )
        END Before;


(* Allocates a block of SZStack bytes on the heap and sets StackBase
   pointing to the start address of the block and TrailBase to the
   end address of the block.

   TrailBase should really be set to [Seg(Ptr^)+SZStack-1:15] so that
   it points to the last byte of the last paragraph in the allocated
   block. However, for some reason this corrupts the heap. So TrailBase
   is set one paragraph lower than the top of block, wasting 16 bytes.
*)

        PROCEDURE OPENStack ();
        VAR Ptr : ADDRESS;
        BEGIN
           (* Can't open an open stack *)
           IF StackOpen THEN RETURN END;

           (* If insufficient memory remains to allocate a stack of SZStack
              bytes, reduce the size of the stack to 2/3 of what is available
           *)

           IF HeapAvail(MainHeap) < SZStack THEN
              SZStack := (HeapAvail(MainHeap) DIV 3) * 2;
           END;

           HeapAllocate (MainHeap,Ptr,SZStack);
           StackBase := Ptr;
           TrailBase := [Seg(Ptr^)+SZStack-2:15];
           TrailTop  := TrailBase;
           NewFrame  := StackBase;
           StackEnd  := StackBase;
           StackTop  := NIL;
           StackOpen := TRUE;
        END OPENStack;


(* Deallocate stack. Stack may not be used until reopened *)

        PROCEDURE CLOSEStack;
        BEGIN
           (* Can't close a closed stack *)
           IF NOT StackOpen THEN RETURN END;
```

```
            HeapDeallocate (MainHeap, StackBase, SZStack);
            StackOpen := FALSE;
        END CLOSEStack;


(* Return the number of bytes between stackend and trailtop *)

        PROCEDURE StackFree () : LONGCARD;
        BEGIN
            RETURN DiffAddr(StackEnd,TrailTop);
        END StackFree;


(* Pushes a record of a variable instantiation onto the trail.
   Returns FALSE if the operation would result in the trail colliding
   with the stack. Else performs operation and returns TRUE.
*)

        PROCEDURE STORETrail   (BPtr : BindPtr) : BOOLEAN;
        VAR TPtr : TrailPtr;
        BEGIN
            IF NOT StackOpen THEN RETURN FALSE; END;
            IF DBG THEN WrStr ('Trail push '); END;
            IF StackFree() < SZTrail THEN
                IF DBG THEN WriteLn ('failed.'); END;
                RETURN FALSE;
            END;
            IF DBG THEN WrAddr(TrailTop); END;
            TrailTop^ := BPtr;
            DecAddr (TrailTop,SZTrail);
            IF DBG THEN WrStr(' -> '); WrAddr(TrailTop); WrLn; END;
            RETURN TRUE;
        END STORETrail;


(* Restore all bindings recorded on the trail from TPtr to trailtop, and
   then pop all trail records from TPtr onwards.
*)

        PROCEDURE RESTORETrail (TPtr : TrailPtr);
        VAR TempPtr : TrailPtr;
        BEGIN
            IF NOT StackOpen THEN RETURN; END;
            TempPtr := TPtr;
            WHILE TempPtr # TrailTop DO
                TempPtr^^.BType := free;
                DecAddr(TempPtr,SZTrail);
            END;
            TrailTop := TPtr;
        END RESTORETrail;


(* Returns the size (in bytes) of a stack frame required to accommodate
   NumVars variable bindings
*)

        PROCEDURE SizeOfFrame (NumVars : CARDINAL) : CARDINAL;
        BEGIN
            RETURN SZFrame0 + NumVars * SZBind;
        END SizeOfFrame;


(* Returns a pointer to a new stack frame immediately below stacktop.
   This is the frame that gets pushed next time PUSHFrame is called.
   The Num, Vars and Trail fields are set, and the variable
   bindings are initialized to free.
```

```
   NIL is returned if creating the frame would cause a collision with
   trailtop.
*)

        PROCEDURE MAKEFrame (Vars : VarIndx) : FramePtr;
        VAR FRMSize : CARDINAL;
        BEGIN
           FRMSize := SizeOfFrame (Vars);
           IF (NOT StackOpen) OR (StackFree() < LONGCARD(FRMSize)) THEN
              RETURN NIL;
           END;
           StackEnd := AddAddr (NewFrame,FRMSize);
           NewFrame^.Vars  := Vars;
           NewFrame^.Trail := TrailTop;

           (* Set frame number to 1 more than previous frame number,
              or to 0 if this is the first frame on the stack. *)
           IF StackTop # NIL THEN
              NewFrame^.Num   := StackTop^.Num + 1;
           ELSE
              NewFrame^.Num   := 0;
           END;

           (* Initialize variable-binding records to free *)
           WHILE Vars # 0 DO
              DEC(Vars);
              NewFrame^.Binds[Vars].BType := free;
           END;

           RETURN NewFrame;
        END MAKEFrame;


(* Pushes NewFrame onto the stack *)

        PROCEDURE PUSHFrame ;
        BEGIN
           (* If stack is closed or there is no NewFrame to push,
              then do nothing
           *)
           IF (NOT StackOpen) OR (NewFrame = StackEnd) THEN
              RETURN;
           END;

           NewFrame^.Prev := StackTop;
           StackTop       := NewFrame;
           NewFrame       := StackEnd;
        END PUSHFrame;


(* Pops all frames on stack from and including FPtr. All variables
   instantiated from this frame onwards are uninstantiated
*)

        PROCEDURE POPFrames (FPtr : FramePtr);
        BEGIN
           IF NOT StackOpen THEN RETURN; END;
           RESTORETrail (FPtr^.Trail);
           StackTop := FPtr^.Prev;
           NewFrame := FPtr;
           StackEnd := FPtr;
        END POPFrames;


(* Return pointer to frame at top of stack *)

        PROCEDURE GetStkTop () : FramePtr;
        BEGIN
           RETURN StackTop;
        END GetStkTop;
```

```
(* ------------------------------------------------------------------------ *)

(* Given a frame ptr and a pointer to a variable, returns pointer to
   binding record of variable within frame.
*)

        PROCEDURE BindAdr (FPtr : FramePtr;
                           TPtr : TermPtr) : BindPtr;
        BEGIN
           RETURN ADR(FPtr^.Binds[TPtr^.Ofst]);
        END BindAdr;


(* Given a pointer to a binding record, returns true if the record
   represents a free variable.
*)

        PROCEDURE IsFree (BPtr : BindPtr) : BOOLEAN;
        BEGIN
           RETURN (BPtr^.BType=free);
        END IsFree;


(* De-reference a variable binding. If the variable is bound to another
   variable, then a pointer to the binding record of the second variable
   is returned. Otherwise the variable is dereferenced to itself.
*)

        PROCEDURE DeRef(BPtr : BindPtr) : BindPtr;
        BEGIN
           IF BPtr^.BType = var THEN RETURN BPtr^.BPtr
           ELSE RETURN BPtr ;
           END;
        END DeRef;


(* Given a TermPtr, return the TermPtr to which it is bound. A non-variable
   term is bound to itself. A variable-term is dereferenced, and the term
   to which it is bound is returned, or a NIL if the variable is free
*)
        PROCEDURE GetBoundTerm (TPtr : TermPtr; FPtr : FramePtr) : TermPtr;
        VAR BPtr : BindPtr;
        BEGIN
           IF IsVar(TPtr) THEN
              BPtr := DeRef(BindAdr(FPtr,TPtr));
              IF IsFree(BPtr) THEN TPtr := NIL
              ELSE TPtr := BPtr^.TPtr;
              END;
           END;
           RETURN TPtr;
        END GetBoundTerm;



(* Returns a pointer to the next literal to be called following
   the literal last called by a frame.
*)

        PROCEDURE NextCall (FPtr : FramePtr) : TermPtr;
        BEGIN
           RETURN FPtr^.CrntLit^.Next;
        END NextCall;



(* ------------------------------------------------------------------------ *)

(* Set stack size to InBytes bytes. The stack size may only be set when the
   stack is closed, and takes effect next time the stack is opened. The
   stack size is rounded up to the nearest whole paragraph (16 bytes), and
```

```
   the OpenStack procedure may override the value set if insufficient
   memory remains on the heap.
*)

        PROCEDURE SetStackSize (InBytes : LONGCARD);
        BEGIN
           IF StackOpen THEN RETURN; END;
           SZStack := CARDINAL ((InBytes + 15) DIV 16);
        END SetStackSize;


(* Report on the stack and trail memory usage. The number of records
   on the trail can be calculated since trail records are of uniform
   size.
*)

        PROCEDURE MEMUsage;
        BEGIN
           WrStr   ('Stack size : '); WrLngCard(LONGCARD(SZStack) * 16,0); WrLn;
           WrStr   ('Stack free : '); WrLngCard (StackFree(),0); WrLn;
           WrStr   ('Trail      : ');
           WrLngCard (DiffAddr (TrailTop,TrailBase) DIV SZTrail,0);
           WriteLn (' records');
           WrStr   ('Stack      : ');
           WrLngCard (DiffAddr (StackBase,StackEnd),0);
           WrStr (' bytes used, ');
           IF StackEnd # NewFrame THEN
              WrCard (NewFrame^.Num+1,0);
           ELSE
              WrCard (StackTop^.Num+1,0);
           END;
           WriteLn (' frames.');
        END MEMUsage;



(* --- Module initialization ---------------------------------------------- *)


BEGIN
   DBG             := FALSE;
   StackOpen       := FALSE;
   SetStackSize (DefaultStkSz);
END Stack.
```

STREAMS - DEFINITION

```
DEFINITION MODULE Streams;

(* This module implements input and output streams.

   Exports:
        Stream (strm) type, EOF condition and line counter.
        Stream redirection procedures.
        Stream I/O routines to substitute those in standard module IO.
*)


TYPE strm = (file, printer, terminal);

VAR PromptStr : ARRAY [1..3] OF CHAR;  (* Prompt when reading from terminal *)

    Line      : CARDINAL;              (* line no. when reading from file *)
    EOF       : BOOLEAN;               (* eof condition when reading from file *)


(* These procedures redirect input and output *)

PROCEDURE ToTerm;
PROCEDURE FromTerm;
PROCEDURE ToPrinter;
PROCEDURE ToFile   (FName : ARRAY OF CHAR);
PROCEDURE FromFile (FName : ARRAY OF CHAR) : BOOLEAN;

PROCEDURE CrntIn  () : strm;
PROCEDURE CrntOut () : strm;


(* (Mostly) substitutes for procedures in module IO *)

PROCEDURE RdStr     (VAR Buffer : ARRAY OF CHAR);
PROCEDURE WrStr     (Str : ARRAY OF CHAR);
PROCEDURE WriteLn   (Str : ARRAY OF CHAR);
PROCEDURE WrLn;
PROCEDURE WrCard    (C : CARDINAL; L : INTEGER);
PROCEDURE WrChar    (C : CHAR);
PROCEDURE WrShtCard (C : SHORTCARD; L : INTEGER);
PROCEDURE WrLngCard (C : LONGCARD; L : INTEGER);
PROCEDURE WrCharRep (C : CHAR; Cnt : CARDINAL);

PROCEDURE WrAddr    (C : ADDRESS);
PROCEDURE ReportErr (m : ARRAY OF CHAR);

PROCEDURE GetKey;
(* Waits for a keypress if current stream is terminal.
   Else returns immediately *)

PROCEDURE RdChar () : CHAR;
(* Reads a character from input (without buffering and echo) if current
   input stream is terminal. Otherwise returns NULL character *)

END Streams.
```

```
IMPLEMENTATION MODULE Streams;

IMPORT FIO;
IMPORT IO;
IMPORT ASCII;

FROM SYSTEM IMPORT Seg, Ofs;

VAR StrmIn, StrmOut : strm;
    In, Out         : FIO.File;


(* ------------------------------------------------------------------------ *)


(* Identify current input and output streams *)

        PROCEDURE CrntIn () : strm;
        BEGIN
           RETURN StrmIn;
        END CrntIn;


        PROCEDURE CrntOut () : strm;
        BEGIN
           RETURN StrmOut;
        END CrntOut;


(* --- Stream Redirecting ---------------------------------------------- *)

        PROCEDURE CloseStrm (s:strm; f:FIO.File);
        BEGIN
           IF s=file THEN FIO.Close(f) END;
        END CloseStrm;


(* Make the terminal the current output stream.
   If the current output stream is a file, then the file is closed
   before the output is directed to the terminal.
*)

      PROCEDURE ToTerm;
        BEGIN
           CloseStrm (StrmOut,Out);
           Out := FIO.StandardOutput;
           StrmOut := terminal;
        END ToTerm;


(* Make the terminal the current input stream.
   If the current input stream is a file, then the file is closed
   before the input stream is directed to the terminal. EOF is
   not accepted from the terminal.
*)

        PROCEDURE FromTerm;
        BEGIN
           EOF := FALSE;
           CloseStrm (StrmIn,In);
           In := FIO.StandardInput;
           StrmIn := terminal;
        END FromTerm;


(* Redirect output to the printer, closing any output file.
*)
```

```
        PROCEDURE ToPrinter;
        BEGIN
           CloseStrm (StrmOut,Out);
           Out := FIO.PrinterDevice;
           StrmOut := printer;
        END ToPrinter;


(*  Redirect output to named file.
    This procedure is not currently used by the program and has
    not been implemented yet.
*)
        PROCEDURE ToFile   (FName : ARRAY OF CHAR);
        BEGIN
        END ToFile;


(* Redirect input from file.
   Returns FALSE if file could not be opened.
   Otherwise closes any input file currently open and opens the named
   file for input, returning TRUE. The line counter is reset to 0, and
   the EOF condition is set to FALSE.
*)

        PROCEDURE FromFile (FName : ARRAY OF CHAR) : BOOLEAN;
        BEGIN
            IF FIO.Exists (FName) THEN
               CloseStrm (StrmIn,In);
               EOF := FALSE;
               Line := 0;
               In := FIO.Open (FName);
               StrmIn := file;
               RETURN TRUE;
            ELSE
               RETURN FALSE;
            END;
        END FromFile;


(* --- Input/Output ------------------------------------------------------ *)


(* Read a string from the current input stream.

   NOTE that if current input is from the terminal, IO.RdStr is used
   instead of FIO.RdStr.  Since IO.RdStr uses the DOS read string (0Ah)
   interrupt 21h, the user gets all the benefits of shell-enhancers such as
   the pd CED, which has history and line editting capabilities.
*)

        PROCEDURE RdStr (VAR Buffer : ARRAY OF CHAR);
        BEGIN
           IF StrmIn = terminal THEN
              IO.WrStr (PromptStr);
              IO.RdStr (Buffer);
           ELSE
              FIO.RdStr (In,Buffer);
              EOF := FIO.EOF;
              INC(Line);
           END;
        END RdStr;



(* The following output routines substitute the routines with the same
   name in the standard IO module.
   The only exception is WriteLn, which is a WrStr followed by a WrLn.
*)

        PROCEDURE WrStr (Str : ARRAY OF CHAR);
```

```
        BEGIN
           FIO.WrStr (Out, Str);
        END WrStr;


        PROCEDURE WriteLn (Str : ARRAY OF CHAR);
        BEGIN
           FIO.WrStr (Out,Str);
           FIO.WrLn  (Out);
        END WriteLn;


        PROCEDURE WrLn;
        BEGIN
           FIO.WrLn (Out);
        END WrLn;


        PROCEDURE WrCard (C : CARDINAL; L : INTEGER);
        BEGIN
           FIO.WrCard (Out,C,L);
        END WrCard;


        PROCEDURE WrChar (C : CHAR);
        BEGIN
           FIO.WrChar (Out,C);
        END WrChar;


        PROCEDURE WrShtCard (C : SHORTCARD; L : INTEGER);
        BEGIN
           FIO.WrShtCard (Out,C,L);
        END WrShtCard;


        PROCEDURE WrLngCard (C : LONGCARD; L : INTEGER);
        BEGIN
           FIO.WrLngCard (Out,C,L);
        END WrLngCard;


        PROCEDURE WrCharRep (C : CHAR; Cnt : CARDINAL);
        BEGIN
           FIO.WrCharRep (Out,C,Cnt);
        END WrCharRep;


(* Report a simple error message *)

        PROCEDURE ReportErr (m:ARRAY OF CHAR);
        BEGIN
           WrLn;WrLn;
           WrStr('*** ERROR: ');
           WriteLn(m);
           WrLn;
        END ReportErr;



(* The following displays an address in Segment:Offset format.
   It is only meant for debugging purposes.
*)

        PROCEDURE WrAddr (C : ADDRESS);
        BEGIN
           WrStr('[');
           WrCard(Seg(C^),0);
           WrStr(':');
           WrCard(Ofs(C^),0);
           WrStr(']');
```

```
        END WrAddr;


(* If the terminal is the current input stream, this procedure waits
   for a key press. Otherwise it does nothing.
   It is used primarily by the debugging code, to give the user time
   to read the carnival-streamer trace.
*)

        PROCEDURE GetKey;
        VAR C : CHAR;
        BEGIN
           IF StrmIn = terminal THEN
              C := IO.RdCharDirect();
           END;
        END GetKey;



(* Reads a character from input (without buffering and echo) if current
   input stream is terminal. Otherwise returns NULL character *)

        PROCEDURE RdChar () : CHAR;
        BEGIN
           IF StrmIn = terminal THEN
              RETURN IO.RdCharDirect();
           ELSE
              RETURN ASCII.nul;
           END;
        END RdChar;



(* --- Module initialization ---------------------------------------------- *)


BEGIN
  FIO.IOcheck := FALSE;
  IO.Prompt   := FALSE;
  PromptStr   := '> ';
  StrmIn  := terminal;
  StrmOut := terminal;
  ToTerm;
  FromTerm;
END Streams.
```

LEX - DEFINITION

```
DEFINITION MODULE Lex;

(* Lexical Analyser.

   Exports.
        The token object.
        Procedures to fetch next token from the input buffer, flush
           input buffer, get current line and column position.

*)

FROM Sstr IMPORT MaxStrLen;

(* ------------------------------------------------------------------------ *)

TYPE TknCls = (ColonHyphen,  (*  :-  *)
               Comma,        (*  ,   *)
               OpnBrk,       (*  (   *)
               ClsBrk,       (*  )   *)
               OpnSqr,       (*  [   *)
               ClsSqr,       (*  ]   *)
               Bar,          (*  |   *)
               Dot,          (*  .   *)
               AnonymVar,    (*  _   *)
               VarSym,       (* UpperCase|_ {UpperCase | LowerCase | Digit } *)
               NonVarSym,    (* LowerCase {UpperCase | LowerCase | Digit }   *)
               Err,          (* unrecognized token *)
               FileEnd       (* End of file on input *)
              );

     Token = RECORD
               Class : TknCls;
               Inst  : ARRAY [1..MaxStrLen] OF CHAR;
             END;


VAR CrntTkn : Token;
    DBG     : BOOLEAN;


PROCEDURE GetToken ;
(* Returns the next token from the input buffer in CrntTkn.
   CrntTkn.Class is set to the TknCls of the fetched token.
   If TknCls is VarSym or NonVarSym, TknInst contains the identifier
      string read in.
*)


PROCEDURE GetPos   (VAR line,char : CARDINAL) ;
(* Returns the line and column position last read from. The line position
   is only meaningful if currently reading from a file. *)


PROCEDURE FlushBuffer;
(* Flush the input buffer *)


PROCEDURE GetItem () : BOOLEAN;
(* Gets next item from the input buffer (skipping any leading spaces) and
   returns it in CrntTkn.Inst.
   Returns TRUE if an item was found, FALSE if at end of line. *)

END Lex.
```

```
IMPLEMENTATION MODULE Lex;

FROM Str     IMPORT CHARSET, Length;
FROM Streams IMPORT Line, EOF, RdStr, WrStr, WriteLn, WrLn, CrntIn, strm;
IMPORT ASCII;


CONST ComChar = '%';            (* comment introducer *)
      MaxBuff = 255;            (* input buffer size *)
      Eof     = ASCII.nul;      (* character to signal end-of-file on input *)

      Separators
              = CHARSET {' ','(',')','.',',','[',']','|',':',"'",ComChar};


VAR Buffer : ARRAY [1..MaxBuff] OF CHAR;  (* Input buffer *)
    BufLen : CARDINAL;                    (* Number of characters in buffer *)
    BufPos : CARDINAL;                    (* Current reading position in buffer *)


(* ------------------------------------------------------------------------ *)


(* Return line and column position.
   The line position is only defined if currently reading from a file,
       and is maintained by the Streams module.
   The column position is the position in the input buffer currently
       being read from, and is defined irrespective of the current input
       stream.
*)

        PROCEDURE GetPos (VAR line,char : CARDINAL) ;
        BEGIN
           line := Line;
           char := BufPos;
        END GetPos;



(* Flush the input buffer.
   This procedure forces procedure GetChar to read in a new line
   the next time it is called to fetch a character from the current
   input stream.
*)

        PROCEDURE FlushBuffer;
        BEGIN
           BufLen := 0;
           BufPos := 0;
        END FlushBuffer;


(* Get next character from the input buffer.
   If the input buffer is exhausted (ie Bufos >= BufLen), then
   a new line is read into the input buffer, and BufPos reset to 0.
   BufPos is incremented, and the character at position BufPos in the
   input buffer is returned.
*)

        PROCEDURE GetChar () : CHAR;
        BEGIN
           IF (BufPos >= BufLen) THEN
           REPEAT
              RdStr (Buffer);
              IF EOF THEN RETURN Eof END;
              BufLen := Length(Buffer);
           UNTIL BufLen > 0;
```

```
            BufPos := 0;
            END;
            INC (BufPos);
            RETURN Buffer[BufPos];
         END GetChar;


(* Unread last character read.
   This procedure is only required by procedure GetToken when reading in an
   identifier. Consequently, it is not a general-purpose unread procedure.
   All it does is to decrement the current BufPos.
*)

         PROCEDURE UnGet;
         BEGIN
            DEC (BufPos);
         END UnGet;


(* A comment introducer has been found. Move the reading position
   past the end of the comment.
   Since Prolog comments are terminated by the end-of-line character,
   all the procedure does is to flush the input buffer, thus forcing
   GetChar to start reading from the next line the next time it is
   called.
*)

         PROCEDURE SkipComment;
         BEGIN
            FlushBuffer;
         END SkipComment;


(* Get the first non-space character from the input buffer.
   Characters are read in until a non-space character is found.
*)

         PROCEDURE GetFirstNonSpace () : CHAR;
         VAR C : CHAR;
         BEGIN
            REPEAT
               C := GetChar();
               IF C=ComChar THEN
                  SkipComment;
                  C := ' ';
               END;
            UNTIL (C # ' ');
            RETURN C;
         END GetFirstNonSpace;


(* Gets next item from the input buffer (skipping any leading spaces) and
   returns it in CrntTkn.Inst.  An item is defined as a sequence of
   non-space characters delimited by a space character or the end-of-line.
   Returns TRUE if an item was found, FALSE if at end of line.
*)

         PROCEDURE GetItem () : BOOLEAN;
         VAR C   : CHAR;
             Cnt : CARDINAL;
         BEGIN
            INC(BufPos);
            WHILE (BufPos <= BufLen) AND (Buffer[BufPos]=' ') DO

               INC(BufPos);
            END;
            IF BufPos > BufLen THEN
```

```
            RETURN FALSE;
         ELSE
            Cnt := 1;
            REPEAT
               CrntTkn.Inst[Cnt] := Buffer[BufPos];
               INC(BufPos);
               INC(Cnt);
            UNTIL (BufPos > BufLen) OR (Buffer[BufPos]=' ');
            CrntTkn.Inst[Cnt] := 0C;
            RETURN TRUE;
         END; (*IF*)
      END GetItem;


(* If in debug mode, produce a wall-paper listing of each token fetched.
*)

      PROCEDURE DEBUG ;
      BEGIN
          IF NOT DBG THEN RETURN END;

          CASE CrntTkn.Class OF
             ColonHyphen  :  WrStr (":-");
          |  Dot          :  WriteLn (".");
          |  Comma        :  WrStr (",");
          |  OpnBrk       :  WrStr ("(");
          |  ClsBrk       :  WrStr (")");
          |  OpnSqr       :  WrStr ("[");
          |  ClsSqr       :  WrStr ("]");
          |  Bar          :  WrStr ("|");
          |  Err          :  WriteLn (" ERROR");
          |  FileEnd      :  WriteLn (" EOF");
          |  VarSym       :  WrStr (" Var:");
                             WrStr (CrntTkn.Inst);
          |  AnonymVar    :  WrStr ('_');
          |  NonVarSym    :  WrStr (" F/P:");
                             WrStr (CrntTkn.Inst);
          END;
      END DEBUG;



(* Returns the next token from the input buffer in CrntTkn.
   CrntTkn.Class is set to the TknCls of the fetched token.
   If TknCls is VarSym or NonVarSym, TknInst contains the identifier
      string read in.
*)

      PROCEDURE GetToken ;
      VAR C : CHAR;

      (* Read in an identifier. The first character has already been read
         in and is passed to the procedure as a parameter. The identifier
         is read into the string CrntTkn.Inst. If the identifier is _ then
         CrntTkn.Class is set to AnonymVar.
      *)

            PROCEDURE GetId (C : CHAR) ;
            VAR Indx   : CARDINAL;
                Quote  : BOOLEAN;
            BEGIN
               Indx := 1;
               IF C="'" THEN
                  Quote := TRUE
               ELSE
                  CrntTkn.Inst[1] := C;
                  Quote := FALSE;
                  INC(Indx);
               END;
               C := GetChar ();
               WHILE ((NOT Quote) AND (NOT (C IN Separators)) OR
```

```
                    (Quote AND (C#"'"))) DO
                    CrntTkn.Inst [Indx] := C;
                    INC (Indx);
                    C := GetChar();
                END;
                CrntTkn.Inst [Indx] := 0C;
                IF Quote THEN RETURN; END;
                IF C # ' ' THEN UnGet END;
                IF (Indx=2) AND (CrntTkn.Inst[1]='_') THEN
                   CrntTkn.Class := AnonymVar;
                END;
             END GetId;

        BEGIN   (* GetToken *)

           C := GetFirstNonSpace ();
           CASE C OF
              Eof : CrntTkn.Class := FileEnd
            | ',' : CrntTkn.Class := Comma
            | '.' : CrntTkn.Class := Dot
            | '(' : CrntTkn.Class := OpnBrk
            | ')' : CrntTkn.Class := ClsBrk
            | '[' : CrntTkn.Class := OpnSqr
            | ']' : CrntTkn.Class := ClsSqr
            | '|' : CrntTkn.Class := Bar
            | ':' : C := GetChar();
                    IF C='-' THEN CrntTkn.Class := ColonHyphen
                    ELSE          CrntTkn.Class := Err;
                    END
            | '_','A'..'Z'
                  : CrntTkn.Class := VarSym;
                    GetId (C)
             ELSE   CrntTkn.Class := NonVarSym;
                    GetId (C)
           END;
           DEBUG ;
        END GetToken;


(* --- module initialization ---------------------------------------------- *)


BEGIN
   CrntTkn.Class := FileEnd;
   CrntTkn.Inst  := '';
   FlushBuffer;
   DBG           := FALSE;
END Lex.
```

PARSE - DEFINITION

```
DEFINITION MODULE Parse;

(* The parser and main interpreter loop.

   Exports:
        ReadInFile, which parses a file.
        Reader, which is the main interpreter loop, reading in and parsing
            user input. The reader first loads the predefined predicates
            from file 'predef.pro', then the file passed as argument (if
            any) - which is the filename given by the user as command-line
            argument, and finally enters the read/parse loop.
*)

VAR DBG : BOOLEAN;

PROCEDURE Reader (FileName : ARRAY OF CHAR);

PROCEDURE ReadInFile (FileName : ARRAY OF CHAR) : BOOLEAN;

END Parse.
```

PARSE - IMPLEMENTATION

```
IMPLEMENTATION MODULE Parse;

(*
    The following CFG is recognised by the parser:
    <program>       ::=   <clause> { <clause> }
    <clause>        ::=   <predicate> [':-' <body>] '.'
    <predicate>     ::=   <predicate symbol>  [ '(' <argument list> ')' ]
    <body>          ::=   <literal> {',' <literal> }
    <literal>       ::=   <variable symbol> |
                          <predicate>
    <argument list> ::=   <term> { ',' <term> }
    <term>          ::=   <variable symbol> |
                          <structure>
    <structure>     ::=   <constant symbol> [ '(' <argument list> ')' ] |
                          <list>
    <list>          ::=   '[' { <term> { ','<term> } [ '|' <term> ] } ']'
    <goal>          ::=   ':-' <body> .

    <predicate symbol> and <constant symbol> are lexically identical
    and are grouped under the class <NonVarSym> by the lexical analyser.

    <literal> and <structure> are syntactically identical, and are parsed
    by the same procedure Literal(). In the symbol table they differ in that
    an entry for a <literal> has a list of clauses associated with it,
    while an entry for a <structure> does not.

    The structure TermRec is used in the representation of both <literals>
    and <terms>.

*)

FROM Streams   IMPORT ToTerm, FromTerm, FromFile, WrStr, WriteLn, WrLn,
                      WrCard, WrShtCard, WrChar, WrCharRep, CrntIn, strm,
                      ReportErr;
FROM Sstr      IMPORT Sptr;
FROM Str       IMPORT Length;
FROM Lex       IMPORT Token, CrntTkn, TknCls, MaxStrLen, GetToken, GetPos,
                      FlushBuffer, GetItem;

FROM DBase     IMPORT TermPtr, TermRec, SymTabPtr, SymTabRec,
                      ClausePtr, ClauseRec, SymType, IsVar, MKClauseRec,
                      MKTermRec, VarIndx, RMTermList, RMClauseRec;
FROM STable    IMPORT Insert, NoCount;

FROM Command   IMPORT ProcessCommand;

FROM Global    IMPORT Exit, Mode, mode;

FROM SYSTEM    IMPORT TSIZE;

FROM ProcGoal  IMPORT ProcessGoal;

FROM Inbuilt   IMPORT DefineInbuilts;


VAR PrsErr   : BOOLEAN;
    Goal     : ClausePtr;


(* ------------------------------------------------------------------------ *)


(* ----------------------- Token.Class Predicate --------------------------
   Checks class of current token.
   IF    (PrsErr) OR (Current token is not of class Class)
         Returns FALSE
   ELSE  Gets next token
```

```
         Returns TRUE
*)

        PROCEDURE Is (Class : TknCls) : BOOLEAN;
        BEGIN
           IF (PrsErr) OR (CrntTkn.Class # Class) THEN RETURN FALSE
           ELSE   GetToken;
                  RETURN TRUE
           END (*IF*);
        END Is;


(* ------------------------ TermRec constructor -----------------------------
   Constructs a new TermRec, and returns a pointer to it. The constructed
   record is initialized as follows

   Entry - as parameter
   Next  - NIL
   SType - as parameter
   If SType is functor,  Args is set to NIL.
   If SType is variable, Ofst is set to the value in the Count field
      of the ST-entry pointed to by Entry.

*)

        PROCEDURE MakeTermRec (Entry : SymTabPtr; SType : SymType) : TermPtr;
        VAR Ptr : TermPtr;
        BEGIN
           Ptr        := MKTermRec (SType);
           Ptr^.Entry := Entry;
           Ptr^.Next  := NIL;
           CASE SType OF
              functor  :  Ptr^.Args := NIL;
           |  variable :  Ptr^.Ofst := Entry^.Count;
           END; (*CASE*)
           RETURN Ptr;
        END MakeTermRec;




(*-------------------------------------------------------------------------*)

    (*
        <literal>  ::=   <variable symbol> | <predicate>
    *)

        PROCEDURE PrsLiteral (VAR Vars : VarIndx) : TermPtr;


            (*
                <term>  ::=   <variable symbol> | <structure>
            *)

             PROCEDURE PrsTerm () : TermPtr;

                 (*
                     <list>  ::=  '[' { <term> {',' <term> } [ '|' <term> ] } ']'
                 *)

                 PROCEDURE PrsList () : TermPtr;
                 VAR FrstPtr, Ptr : TermPtr;
                 BEGIN
                    FrstPtr := MakeTermRec (NIL, list);
                    Ptr     := FrstPtr;
                    IF NOT Is(ClsSqr) THEN
                       REPEAT
                       Ptr^.Args := PrsTerm();
                       Ptr       := Ptr^.Args;
                       IF (NOT PrsErr) AND
```

```
                        ((CrntTkn.Class = Comma) OR (CrntTkn.Class = ClsSqr))
THEN
                        Ptr^.Next := MakeTermRec(NIL,list);
                        Ptr := Ptr^.Next;
                     END; (*IF*)
                     UNTIL NOT Is (Comma);
                     CASE CrntTkn.Class OF
                        ClsSqr : Ptr^.Args := NIL;
                                 GetToken;
                     |  Bar    : GetToken;
                                 Ptr^.Next := PrsTerm();
                                 IF NOT Is(ClsSqr) THEN PrsErr := TRUE END;
                        ELSE PrsErr := TRUE;
                     END; (*CASE*)
                  ELSE
                     Ptr^.Args := NIL;
                  END; (*IF*)
                  RETURN FrstPtr;
               END PrsList;


            VAR TPtr : TermPtr;
                SPtr : SymTabPtr;

            BEGIN
            CASE CrntTkn.Class OF
               VarSym    : SPtr := Insert(CrntTkn.Inst,variable,0);
                           IF SPtr^.Count = NoCount THEN
                              SPtr^.Count := Vars;
                              INC(Vars);
                           END; (*IF*)
                           TPtr := MakeTermRec (SPtr,variable);
                           GetToken;
            |  NonVarSym : TPtr := PrsLiteral (Vars);
            |  AnonymVar : TPtr := MakeTermRec (NIL,anon);
                           GetToken;
            |  OpnSqr    : GetToken;
                           TPtr   := PrsList();
               ELSE        TPtr   := NIL;
                           PrsErr := TRUE;
            END; (*CASE*)
            RETURN TPtr;
            END PrsTerm;


            (*
                <argument list> ::= <term> {, <term> }
            *)

            PROCEDURE PrsArgList (VAR Arity : SHORTCARD) : TermPtr;
            VAR FrstPtr, Ptr : TermPtr;
            BEGIN
               FrstPtr := PrsTerm();
               Ptr     := FrstPtr;
               INC(Arity);
               WHILE Is(Comma) DO
                  Ptr^.Next := PrsTerm ();
                  Ptr       := Ptr^.Next;
                  INC (Arity);
               END;
               RETURN FrstPtr;
            END PrsArgList;


         VAR Arity : SHORTCARD;
             Tkn   : Token;
             Ptr   : TermPtr;

         BEGIN
            Arity  := 0;
            CASE CrntTkn.Class OF
```

```
               NonVarSym     :  Tkn  := CrntTkn;
                                Ptr  := MakeTermRec(NIL,functor);
                                GetToken;
                                IF Is (OpnBrk) THEN
                                   Ptr^.Args := PrsArgList(Arity);
                                   IF NOT Is (ClsBrk) THEN PrsErr := TRUE; END;
                                END; (*IF*)
                                IF NOT PrsErr THEN
                                   Ptr^.Entry := Insert(Tkn.Inst,functor,Arity);
                                END;
            |    VarSym        :  Ptr  := PrsTerm ();
                 ELSE              PrsErr := TRUE;
                                   Ptr    := NIL;
            END; (*CASE*)
            IF DBG THEN
               WrStr   (" /");
               WrShtCard  (Arity,0);
            END;
            RETURN Ptr;
         END PrsLiteral;



(*
    <predicate>     ::=   <predicate symbol>  [ '(' <argument list> ')' ]
*)

         PROCEDURE PrsPred (VAR Vars : VarIndx) : TermPtr;
         BEGIN
            IF (CrntTkn.Class = NonVarSym) THEN
               RETURN PrsLiteral (Vars);
            ELSE
               PrsErr := TRUE;
               RETURN NIL;
            END; (*IF*)
         END PrsPred;


(*
    <body>          ::=   <literal> {',' <literal> }
*)

         PROCEDURE PrsBody (VAR Vars : VarIndx) : TermPtr;
         VAR TPtr : TermPtr;
             Ptr  : TermPtr;
         BEGIN
            TPtr := PrsLiteral(Vars);
            Ptr  := TPtr;
            WHILE Is (Comma) DO
               Ptr^.Next := PrsLiteral(Vars);
               Ptr       := Ptr^.Next;
            END;
            RETURN TPtr;
         END PrsBody;


(* Parse a clause. This parses

     1. definite clauses (assertions and rules).
     2. goal clauses (introduced by :- ).
     3. commands (introduced by |).

   Goal clauses are not inserted in the database, but are passed to the
   ProcessGoal procedure in module ProcGoal.

   Commands are passed to the ProcessCommand procedure.

*)

         PROCEDURE PrsClause;
         VAR  CPtr   : ClausePtr;
```

```
      SPtr   : SymTabPtr;
      NVars  : VarIndx;
      IsGoal : BOOLEAN;


  (* Procedure to reset the Count fields of ST entries to NoCount
     after a clause has been compiled
  *)

      PROCEDURE ResetVarCounts (TPtr : TermPtr);
      BEGIN
         WHILE (TPtr # NIL) DO
           CASE TPtr^.SType OF
              variable : TPtr^.Entry^.Count := NoCount;
           |  list,functor  : ResetVarCounts(TPtr^.Args);
           END; (*CASE*)
           TPtr := TPtr^.Next;
         END; (*WHILE*)
      END ResetVarCounts;


 BEGIN

    IF DBG THEN WrLn END;
    IsGoal := FALSE;

    (* Command? *)
    IF CrntTkn.Class = Dot THEN
       PrsErr := NOT ProcessCommand ();
       FlushBuffer;
       IF (NOT PrsErr) AND (CrntIn()=file) THEN
          GetToken;
       END;
       RETURN;
    END;

 (* Initialize variables counter. The counter is incremented with
    every new variable encountered in the clause by the Literal
    parsing routine.
 *)


    NVars := 0;


 (* Allocate a clause record from the heap and set the Head field
    pointing to the literal at the head of the clause. If a :- token
    is found, then set the Body field to the list of literals making
    up the body of the clause, else the Body is NIL.
 *)

    IF Is (ColonHyphen) THEN
       IsGoal := TRUE;
       CPtr := Goal;
       CPtr^.Head := NIL;
       CPtr^.Body := PrsBody(NVars);
    ELSE
       IsGoal := FALSE;
       CPtr := MKClauseRec ();
       CPtr^.InBlt := FALSE;
       CPtr^.Next  := NIL;
       CPtr^.Body  := NIL;
       CPtr^.Head  := PrsPred(NVars);

   (* Attempt to redefine a system-defined predicate by user ? *)

       IF (Mode=user) AND (CPtr^.Head^.Entry^.Mode=system) THEN
          PrsErr := TRUE;
       ELSIF Is (ColonHyphen) THEN
          CPtr^.Body   := PrsBody(NVars);
       END; (*IF*)
```

```
            END; (*IF*)

          (* Reset all the Count fields of ST-entries for variables in this
             clause to NoCount.
          *)


              ResetVarCounts (CPtr^.Head);
              ResetVarCounts (CPtr^.Body);

         (* IF no error was reported while parsing THEN
            a. if the clause is a definite clause, then link it into the
               clause database. The new clause is to be linked to the end of the list
               of clauses for this predicate, pointed to by the literal pointed
               to by the Head field.
            b. if a goal, then pass it to ProcessGoal.
         *)

             IF (CrntTkn.Class = Dot) AND NOT PrsErr THEN
                IF CrntIn() = file THEN
                   GetToken;
                END;
                CPtr^.Vars := NVars;
                IF NOT IsGoal THEN
                   SPtr := CPtr^.Head^.Entry;
                   IF   SPtr^.FstCls  = NIL THEN
                        SPtr^.FstCls := CPtr
                   ELSE SPtr^.LstCls^.Next := CPtr;
                   END; (*IF*)
                   SPtr^.LstCls      := CPtr;
                ELSE
                   ProcessGoal (CPtr);
                   RMTermList (Goal^.Body);
                END; (*IF*)

             ELSE

            (* An error has occured. If parsing a goal, then deallocate the
               body, else deallocate the whole clause.
            *)

                PrsErr := TRUE;
                IF IsGoal THEN RMTermList (Goal^.Body)
                ELSE RMClauseRec (CPtr);
                END;

             END; (*IF*)

          END PrsClause;


(* Parses a file.
   Returns TRUE if file parsed correctly.
   Otherwise outputs an error message (together with the line and column
   position of the error in the file, if applicable), and returns FALSE.
*)

          PROCEDURE ReadInFile (FileName : ARRAY OF CHAR) : BOOLEAN;
          VAR ok  : BOOLEAN;
              L,C : CARDINAL;
          BEGIN
             ok := (FromFile (FileName));
             IF ok THEN
                GetToken;
                WHILE NOT (PrsErr) AND (CrntTkn.Class # FileEnd) DO
                   PrsClause;
                END; (*WHILE*)
                IF PrsErr THEN
                   GetPos(L,C);
                   WrStr  ('Error in ');
                   WrStr  (FileName);
```

```
                WrStr  (' at line ');
                WrCard (L,0);
                WrStr  (' column ');
                WrCard (C,0);
                WrLn;
                ok := FALSE;
             END;
          ELSE
             WrStr (FileName);
             WrStr (' not found.');
             WrLn;
          END;
          FromTerm;
          PrsErr := FALSE;
          RETURN ok;
       END ReadInFile;


(* This is the main loop of the interpreter.
   It reads in user input and passes it to the parser until the Exit
   flag becomes TRUE.

   The reader does the following:
   1.   Reads in the file 'predef.pro' containing the predefined
        predicates.
   2.   Reads in the file (if any) whose name is passed as an
        input parameter. This is the command-line argument given
        by the user when the interpreter is invoked at the DOS prompt.
   3.   Enters a loop - get input, parse input.
*)

       PROCEDURE Reader (FileName : ARRAY OF CHAR);
       VAR L,P : CARDINAL;
       BEGIN
          PrsErr := FALSE;
          Goal   := MKClauseRec();
          Goal^.InBlt := FALSE;
          Goal^.Next := NIL;
          IF NOT ReadInFile('predef.pro') THEN
             ReportErr ('Bad or missing PREDEF.PRO');
             RETURN
          END;

          Mode := user;

          IF Length (FileName) # 0 THEN
             IF ReadInFile(FileName) THEN END;
          END;

          REPEAT
             FlushBuffer;
             PrsErr := FALSE;
             GetToken;
             PrsClause;
             IF PrsErr THEN
                ReportErr ('Parse error in user input');
             END;
          UNTIL Exit;

       END Reader;


(* --- Module initialization ---------------------------------------------- *)


BEGIN
   DBG := FALSE;
END Parse.
```

```
DEFINITION MODULE Command;

(* Command processor.
   Processes user commands (introduced by a '.')

   Exports:
        the command processor.

*)


(* Command processor: takes input directly from the input buffer.
   Returns FALSE to indicate an error.
           TRUE  otherwise.
   Flushes input buffer on exit.
*)

PROCEDURE ProcessCommand() : BOOLEAN;

END Command.
```

```
IMPLEMENTATION MODULE Command;

(* Commands implemented
   .LIST  - list database
   .LOAD  - load a file
   .STATS - display memory usage statistics
   .DEBUG - selectively toggle debugging switches
   .STACK - set stack size (in bytes)
   .EXIT  - exit program
*)

FROM Lex     IMPORT CrntTkn, GetItem;
FROM STable  IMPORT ListDBase;
FROM Parse   IMPORT ReadInFile;
FROM Str     IMPORT Caps, Compare, StrToCard;
FROM Streams IMPORT WrStr, WriteLn, WrLngCard, WrLn, RdChar;
FROM Storage IMPORT HeapAvail, HeapTotalAvail, MainHeap;
FROM Global  IMPORT Exit;

IMPORT Stack;
IMPORT DBase;
IMPORT Lex;
IMPORT STable;
IMPORT Parse;
IMPORT ProcGoal;

(* ------------------------------------------------------------------------ *)

        (* Support procedure for printing out DEBUG switch
           settings.
        *)

        PROCEDURE WrDBG (S : ARRAY OF CHAR; B : BOOLEAN);
        BEGIN
           WrStr(S);
           IF B THEN
              WriteLn (' - ON');
           ELSE
              WriteLn (' - OFF');
           END;
        END WrDBG;


(* ------------------------------------------------------------------------ *)


        PROCEDURE ProcessCommand () : BOOLEAN;
        VAR ok : BOOLEAN;
            C  : LONGCARD;

        BEGIN
           ok := TRUE;

           IF GetItem() THEN
              Caps(CrntTkn.Inst);

              IF     Compare (CrntTkn.Inst,"LIST")=0 THEN
                     ListDBase;

              ELSIF  Compare (CrntTkn.Inst,"LOAD")=0 THEN
                     IF GetItem() THEN
                        ok :=  ReadInFile (CrntTkn.Inst);
                     ELSE
                        WriteLn ("Filename expected.");
                        ok := FALSE;
                     END;
```

```
            ELSIF  Compare (CrntTkn.Inst, "DEBUG")=0 THEN
                      WrDBG ('1: Lexical analyser',Lex.DBG);
                      WrDBG ('2: Syntax  analyser',Parse.DBG);
                      WrDBG ('3: Symbol table    ',STable.DBG);
                      WrDBG ('4: Interpreter     ',ProcGoal.DBG);
                      WrDBG ('5: Stack and trail ',Stack.DBG);
                      CASE RdChar() OF
                           '1' : Lex.DBG := NOT Lex.DBG;
                      |    '2' : Parse.DBG := NOT Parse.DBG;
                      |    '3' : STable.DBG := NOT STable.DBG;
                      |    '4' : ProcGoal.DBG := NOT ProcGoal.DBG;
                      |    '5' : Stack.DBG := NOT Stack.DBG;
                      END; (*CASE*)

            ELSIF  Compare (CrntTkn.Inst, "STACK")=0 THEN
                      IF GetItem() THEN
                         C := StrToCard (CrntTkn.Inst,10,ok);
                         IF ok THEN
                            Stack.SetStackSize(C);
                         END;
                      ELSE ok := FALSE;
                      END;
                      IF NOT ok THEN
                         WriteLn('Size in bytes expected.');
                      END;

            ELSIF  Compare (CrntTkn.Inst, "EXIT")=0 THEN
                      Exit := TRUE;

            ELSE ok := FALSE;
            END;

         ELSE ok := FALSE;
         END;

         RETURN ok;
      END ProcessCommand;

END Command.
```

```
DEFINITION MODULE ProcGoal;

(* The interpreter.

   Exports:
        The interpreter ProcessGoal.
        A (runtime) error indicator.

*)


FROM DBase IMPORT ClausePtr;

VAR DBG : BOOLEAN;
    Err : BOOLEAN;

PROCEDURE ProcessGoal (Goal : ClausePtr);

END ProcGoal.
```

```
IMPLEMENTATION MODULE ProcGoal;

FROM Stack   IMPORT Frame, FramePtr, MAKEFrame, PUSHFrame,
                    BindPtr, BindType, OPENStack, CLOSEStack,
                    STORETrail, RESTORETrail, POPFrames, Before,
                    DeRef, BindAdr, GetBoundTerm, IsFree, NextCall, GetStkTop;

FROM DBase  IMPORT  ClauseRec, TermRec, TermPtr, SymTabRec, SymTabPtr, IsVar,
                    GetFunctor, IsFunctor, IsAssertion, IsAnon,
                    IsList, IsNulList, IsNonNulList, SameFunctor,
                    VarIndx;

FROM STable  IMPORT ListTerm, ListClause, SymType, Test, order;

FROM Streams IMPORT WrStr, WrLn, WrCard, WrCharRep, WrShtCard, WriteLn,
                    GetKey, ReportErr;
FROM Storage IMPORT HeapTotalAvail, MainHeap;
FROM Inbuilt IMPORT InBltProc;
FROM Global  IMPORT MEMUsage;


(* ------------------------------------------------------------------------ *)

    (* Lists out a term instance. This is analogous to ListTerm in module
       STable, except that it outputs a constructed term using the variable
       bindings.
    *)

    PROCEDURE ListBTerm (TPtr : TermPtr; FPtr : FramePtr);
    VAR BPtr : BindPtr;

         PROCEDURE ListArgs (TPtr : TermPtr; FPtr : FramePtr);
         BEGIN
            TPtr := TPtr^.Args;
            IF TPtr=NIL THEN RETURN END;
            WrStr('(');
            WHILE TPtr#NIL DO
               ListBTerm(TPtr, FPtr);
               TPtr := TPtr^.Next;
               IF TPtr#NIL THEN WrStr(',') END;
            END;
            WrStr(')');
         END ListArgs;

    BEGIN
       CASE TPtr^.SType OF
          variable : BPtr := DeRef(BindAdr(FPtr,TPtr));
                     IF IsFree(BPtr) THEN WrStr('*')
                     ELSE
                       IF DBG THEN
                          WrStr('{'); WrCard(BPtr^.Env^.Num,0); WrStr('} ');
                       END;
                       ListBTerm (BPtr^.TPtr, BPtr^.Env);
                     END;
       |  functor  : WrStr(TPtr^.Entry^.Name^);
                     ListArgs (TPtr,FPtr);
       |  anon     : WrStr ('_');
       |  list     : WrStr('[');
                     WHILE IsNonNulList(TPtr) DO
                        TPtr := TPtr^.Args;
                        ListBTerm (TPtr,FPtr);
                        TPtr := TPtr^.Next;
                        IF IsVar(TPtr) THEN
                           BPtr := DeRef(BindAdr(FPtr,TPtr));
                           TPtr := BPtr^.TPtr;
                           FPtr := BPtr^.Env;
                        END; (*IF*)
```

```
                        IF IsNonNulList (TPtr) THEN
                           WrStr (',');
                        END; (*IF*)
                     END; (*WHILE*)
                     IF IsNulList(TPtr) THEN
                        WrStr (']')
                     ELSE
                        WrStr('|');
                        ListBTerm (TPtr,FPtr);
                        WrStr(']');
                     END; (*IF*)
      END; (*CASE*)
   END ListBTerm;


(* Prints out all the bindings of the variables in a clause instance.

   INPUT   Clause - pointer to clause prototype;
           Frame  - pointer containing bindings for this instance;

   OUTPUT  Displays the bindings of all the variables in the clause
           in the form <var> = <binding>. If clause contains no
           variables, outputs 'YES'.

   Since the representation of a clause prototype does not contain a
   list of the variables in the clause, the procedure has to traverse
   the prototype looking for each variable in turn. The traversal is
   performed by the auxiliary procedure FindVar. The bindings are output
   by the procedure ListTerm.
*)

PROCEDURE OutPutBindings (Clause : ClausePtr; Frame : FramePtr);
VAR Dummy : VarIndx;

   PROCEDURE FindVar (TPtr : TermPtr; Num : VarIndx) : VarIndx;
   BEGIN
      WHILE (TPtr # NIL) DO
         IF IsFunctor (TPtr) OR IsList(TPtr) THEN
            Num := FindVar (TPtr^.Args,Num)
         ELSIF (IsVar(TPtr) AND (TPtr^.Ofst = Num)) THEN
            WrStr ( '   ');
            WrStr ( TPtr^.Entry^.Name^);
            WrStr ( ' = ');
            ListBTerm (TPtr, Frame);
            WrLn  ;
            INC (Num);
         END;
         TPtr := TPtr^.Next;
      END; (*WHILE*)
      RETURN Num;
   END FindVar;

BEGIN
   IF (Clause^.Vars # 0) THEN
      Dummy := FindVar (Clause^.Body, FindVar(Clause^.Head,0));
   END;
END OutPutBindings;



(* ---- debugging ------------------------------------------------------ *)

   (* Wallpaper-dump of stackframe pointed to by FPtr. Logically the
      procedure should belong to module Stack, but because it requires
      ListTerm to list the variable bindings recorded in the frame
      it was shifted to this module
   *)

   PROCEDURE DumpFrame (FPtr : FramePtr);
   VAR J : SHORTCARD;
   BEGIN
```

```
        IF NOT DBG THEN RETURN END;
          WrCharRep ('-',79);
          WrLn      ;
          WrStr     ('FRAME     : ') ; WrCard    (FPtr^.Num,0)  ; WrLn ;
          WrStr     ('VARS      : ') ; WrCard    (FPtr^.Vars,0) ; WrLn ;
          WrStr     ('PARENT    : ') ;
          IF (FPtr^.Parent = NIL) THEN
             WrStr  ('None');
          ELSE
             WrStr  ('Frame #');
             WrCard (FPtr^.Parent^.Num,0);
          END;
          WrLn  ;
          WrStr ('CrntLit   : ');
          IF (FPtr^.CrntLit = NIL) THEN
               WrStr ('-'); WrLn;
          ELSE ListTerm (FPtr^.CrntLit); WrLn;
          END;
          WrStr ('CrntCls   : ');
          ListClause (FPtr^.CrntCls);
          WrStr ('NxtClause : ');
          IF (FPtr^.NxtClause = NIL) THEN WrStr('-')
          ELSE ListClause (FPtr^.NxtClause)
          END;
          WrLn ;
          OutPutBindings (FPtr^.CrntCls,FPtr);
          WrLn ; WrLn ;
      END DumpFrame;


(* ------------------------------------------------------------------------ *)

(* This is the main interpreter routine.

   This routine should really be a module by itself,  with the procedures
   above forming a separate support module.

   The procedure is divided into 5 main sections

   1.   initialization
   2.   main control loop
   3.   procedure selection and unification
   4.   backtracking
   5.   processing of inbuilt predicates

*)

PROCEDURE ProcessGoal (Goal : ClausePtr);
VAR Root         : FramePtr;
    Parent       : FramePtr;
    BKTrackPoint : FramePtr;
    NewFrame     : FramePtr;
    CrntCall     : TermPtr;
    CrntProc     : ClausePtr;
    Solutions    : CARDINAL;
    BPtr         : BindPtr;


    (* --- inbuilt predicates ---------------------------------------- *)


    PROCEDURE ExecInBlt (Proc : InBltProc) : BOOLEAN;

          PROCEDURE TestLex(Ord : order) : BOOLEAN;
          VAR T1,T2 : TermPtr;
              B     : BindPtr;
          BEGIN
             T1 := GetBoundTerm(CrntCall^.Args,Parent);
             T2 := GetBoundTerm(CrntCall^.Args^.Next,Parent);
             IF (T1=NIL) OR (T2=NIL) THEN RETURN FALSE END;
             IF NOT((T1^.SType = functor) AND (T2^.SType = functor)) THEN
```

```
            RETURN FALSE
          ELSE RETURN Test(T1^.Entry,T2^.Entry,Ord);
          END;
        END TestLex;


VAR ok : BOOLEAN;
BEGIN
    ok := TRUE;
    CASE Proc OF
         cut     : BKTrackPoint := Parent^.CrntBTP;
      |  isvar   : ok := IsVar  (CrntCall^.Args) AND
                         IsFree (DeRef(BindAdr(Parent,CrntCall^.Args)));
      |  nl      : WrLn;
      |  write   : ListBTerm(CrntCall^.Args,Parent);
      |  fail    : ok := FALSE;
      |  stats   : MEMUsage;
      |  lexlt   : ok := TestLex(lt);
      |  lexgt   : ok := TestLex(gt);
      |  lexle   : ok := TestLex(le);
      |  lexge   : ok := TestLex(ge);
    END;
    IF ok THEN CrntCall := CrntCall^.Next; END;
    RETURN ok;
END ExecInBlt;



(* --- unification ------------------------------------------------ *)


 PROCEDURE UnifyTerm (LP1, LP2 : TermPtr;
                      EN1, EN2 : FramePtr) : BOOLEAN;

 VAR BPtr1, BPtr2 : BindPtr;
     Var1,  Var2  : BOOLEAN;
     Success      : BOOLEAN;

 BEGIN
    IF Err THEN RETURN FALSE; END;
    IF DBG THEN
       WrStr('UNIFYING: '); WrStr('{'); WrCard(EN1^.Num,0); WrStr('} ');
                               ListTerm (LP1); WrLn;
       WrStr('AND     : '); WrStr('{'); WrCard(EN2^.Num,0); WrStr('} ');
                               ListTerm (LP2); WrLn;
       GetKey;
    END;

    Success := TRUE;
    IF (IsAnon(LP1) OR IsAnon(LP2)) THEN RETURN Success END;
    Var1    := IsVar(LP1);
    Var2    := IsVar(LP2);
    IF Var1 THEN BPtr1 := DeRef(BindAdr(EN1,LP1)) END;
    IF Var2 THEN BPtr2 := DeRef(BindAdr(EN2,LP2)) END;

    IF NOT (Var1 OR Var2) THEN
       Success := SameFunctor(LP1,LP2);
       LP1     := LP1^.Args;
       LP2     := LP2^.Args;
       WHILE (Success) AND (LP1 # NIL) DO
          Success := UnifyTerm (LP1,LP2,EN1,EN2);
          LP1     := LP1^.Next;
          LP2     := LP2^.Next;
       END; (*WHILE*);

    (* At least one of LP1 and LP2 must be variables.
       This leads to the following cases:

       1. var LP1, var LP2
          let D1 and D2 be the dereferences of LP1 and LP2 respectively.
          If D1 = D2 then succeed (since both are instantiated to the
```

```
            same thing).
         If both D1 and D2 are bound, then UnifyTerm Lit(D1), Lit(D2)
            with environments Env(D1), Env(D2).
         If D2 is free then bind D2 to D1.
         If D1 is free then copy D2 to D1 and set trail

      2. literal LP1, var LP2
         let D2 be the dereference of LP2.
         if D2 is free then bind D2 to LP1 with EN1 as environment
         else UnifyTerm LP1, Lit(D2) with environments EN1, Env(D2)

      3. var LP1, literal LP2
         let D1 be the dereference of LP1
         if D1 is free then bind D1 to LP2 with EN2 as environment
         else UnifyTerm Lit(D1), LP2 with environments Env(D1), EN2
   *)

   ELSIF (Var1 AND Var2) THEN
      IF BPtr1 # BPtr2 THEN
         IF NOT (IsFree(BPtr1) OR IsFree(BPtr2)) THEN
            Success := UnifyTerm (BPtr1^.TPtr, BPtr2^.TPtr,
                                  BPtr1^.Env,  BPtr2^.Env);
         ELSIF IsFree(BPtr2) THEN
               BPtr2^.BType := var;
               BPtr2^.BPtr := BPtr1;
               IF Before(BPtr2,BKTrackPoint) THEN
                  Err := NOT STORETrail (BPtr2);
               END;
         ELSE  BPtr1^ := BPtr2^;
               IF Before(BPtr1,BKTrackPoint) THEN
                  Err := NOT STORETrail (BPtr1)
               END;
         END;
      END;

   ELSIF (Var2) THEN
      IF IsFree(BPtr2) THEN
         BPtr2^.BType     := lit;
         BPtr2^.TPtr      := LP1;
         BPtr2^.Env       := EN1;
         IF Before(BPtr2,BKTrackPoint) THEN
            Err := NOT STORETrail (BPtr2);
         END;
      ELSE
         Success := UnifyTerm (LP1, BPtr2^.TPtr,
                               EN1, BPtr2^.Env);
      END;

   ELSIF (Var1) THEN
      IF IsFree(BPtr1) THEN
         BPtr1^.BType     := lit;
         BPtr1^.TPtr      := LP2;
         BPtr1^.Env       := EN2;
         IF Before(BPtr1,BKTrackPoint) THEN
            Err := NOT STORETrail (BPtr1);
         END;
      ELSE
         Success := UnifyTerm (BPtr1^.TPtr, LP2,
                               BPtr1^.Env,  EN2);
      END;
   END; (*IF*)
   RETURN Success;
END UnifyTerm;

(* Attempts to resolve CrntCall with the head of a clause starting
   from CrntProc.
   INPUTS   - none;
   OUTPUTS  - exits with CrntProc pointing to the clause whose head
              successfully resolved with the CrntLit, or NIL if no
              clause responded to the call. NewFrame contains any
              bindings for the variables in the clause CrntProc.
```

```
*)

PROCEDURE SelectProc () : BOOLEAN;
BEGIN
   IF (CrntProc # NIL) AND (CrntProc^.InBlt) THEN
      RETURN ExecInBlt(CrntProc^.Proc);
   END;
   WHILE (CrntProc # NIL) AND NOT Err DO
      NewFrame := MAKEFrame(CrntProc^.Vars);
      IF NewFrame # NIL THEN

         (* Is this a backtrack point ? *)

         NewFrame^.CrntBTP := BKTrackPoint;
         IF CrntProc^.Next # NIL THEN
            BKTrackPoint := NewFrame;
         END;

         IF UnifyTerm(CrntCall,CrntProc^.Head,Parent,NewFrame) THEN

            (* Complete activation record and push it *)

            NewFrame^.Parent    := Parent;
            NewFrame^.CrntLit   := CrntCall;
            NewFrame^.CrntCls   := CrntProc;
            NewFrame^.NxtClause := CrntProc^.Next;
            Parent              := NewFrame;
            PUSHFrame();
            CrntCall            := CrntProc^.Body;
            RETURN TRUE;

         ELSE

            (* Backtrack one frame - shallow backtracking *)

            RESTORETrail (NewFrame^.Trail);
            CrntProc := CrntProc^.Next;
            BKTrackPoint := NewFrame^.CrntBTP;

         END; (*IF*)

      ELSE Err := TRUE;
      END; (*IF*)

   END; (*WHILE*)
   RETURN FALSE;
END SelectProc;



(* --- backtracking -----------------------------------------------
   Deep backtracking - resume interpretation from the call indicated
        by the current BKTrackPoint, if any.
*)

PROCEDURE BackTrack () : BOOLEAN;
VAR NewBTP : FramePtr;
BEGIN
   IF DBG THEN
      WrStr('Backtracking ');
   END;
   IF BKTrackPoint = NIL THEN
      IF DBG THEN
         WrStr(' failed');
         WrLn ;
      END;
      RETURN FALSE;
   ELSE
      IF DBG THEN
         WrStr ('to : ');
         WrCard(BKTrackPoint^.Num,0);
```

```
         WrLn ;
      END;
      CrntProc     := BKTrackPoint^.NxtClause;
      CrntCall     := BKTrackPoint^.CrntLit;
      Parent       := BKTrackPoint^.Parent;
      NewBTP       := BKTrackPoint^.CrntBTP;
      POPFrames    (BKTrackPoint);
      BKTrackPoint := NewBTP;
      RETURN TRUE;
   END;
END BackTrack;


(* --- main control loop ------------------------------------------ *)


PROCEDURE Run;
BEGIN
LOOP

(* Quit if an error has occured *)

   IF Err THEN RETURN; END;

(* Select Call:
   CrntProc points to clause selected by previous call to SelectProc.

   IF current procedure is an assertion (ie has no calls) then we have
      arrived at a leaf of the search tree, so go up to the first parent
      procedure which still has some calls pending.

      IF no parent has pending calls, then we have found a solution.
      Output solution and force backtracking to search for further
      solutions.

   OTHERWISE prepare to enter the procedure CrntProc at its first
             call.

*)

    WHILE (CrntCall=NIL) AND (Parent # Root) DO
         CrntCall := NextCall(Parent);
         Parent   := Parent^.Parent;
    END;


   (* CrntCall is NIL iff no parent node has pending calls. In this
      case we have found a solution, and backtracking is required
      to search for further solutions. If backtracking fails, then
      quit. Note that inbuilt predicates are never backtracked to since
      all such predicates are deterministic.
   *)

   IF CrntCall = NIL THEN
      INC (Solutions);
      IF Goal^.Vars > 0 THEN
         WrStr  ( '-- solution ');
         WrCard ( Solutions,0);
         WrLn;
         OutPutBindings (Goal, Root);
      END; (*IF*)
      IF NOT BackTrack() THEN RETURN; END;

   ELSE

   (* Otherwise, CrntCall points to the next call to be executed.
      The call may be a variable, in which case the literal
      dereferenced by the variable is the actual call. If the variable
      is unbound, or bound to a free variable, then an error occurs.
   *)
```

```
            IF IsVar(CrntCall) THEN
               BPtr := DeRef(BindAdr(Parent,CrntCall));
               IF IsFree(BPtr) THEN
                  Err := TRUE;
               ELSE
                  CrntProc := BPtr^.TPtr^.Entry^.FstCls;
               END;
            ELSIF IsFunctor (CrntCall) THEN
               CrntProc    := CrntCall^.Entry^.FstCls;
            END;
         END;

      (* Select procedure to respond to this call. If no procedure responds,
         then we have met a failure node, so backtracking is required. If
         backtracking fails, then quit.
      *)

         WHILE NOT SelectProc() DO
            IF Err THEN RETURN; END;
            IF NOT BackTrack() THEN RETURN; END;
         END; (*LOOP*)

         DumpFrame (GetStkTop());

      END; (*LOOP*)
      END Run;

(* --- control initialization ------------------------------------------- *)


BEGIN
   Err := FALSE;
   OPENStack;
   Solutions       := 0;
   BKTrackPoint    := NIL;
   Root            := MAKEFrame(Goal^.Vars);
   Root^.CrntBTP   := NIL;
   Root^.CrntLit   := NIL;
   Root^.NxtClause := NIL;
   Root^.CrntCls   := Goal;
   Root^.Parent    := NIL;
   Parent          := Root;
   CrntCall        := Goal^.Body;
   PUSHFrame;
   DumpFrame (Root);
   Run;

   IF NOT Err THEN
      IF (Solutions=0) THEN
         WrStr ('NO');
         WrLn;
      ELSIF (Goal^.Vars = 0) THEN
         WrStr ('YES');
         WrLn ;
      END; (*IF*)
   ELSE
      ReportErr ('Run-time error');
   END;

   CLOSEStack;

END ProcessGoal;



(* ---- module initialization ------------------------------------------- *)


BEGIN
  DBG := FALSE;
END ProcGoal.
```

INBUILT - DEFINITION

```
DEFINITION MODULE Inbuilt;

(* Definition of inbuilt predicates - interface module.

   Exports:
        inbuilt-predicate identifiers.
        definition procedure.

*)

TYPE InBltProc = (isvar, cut, write, nl, fail, stats,
                  lexlt, lexgt, lexle, lexge);

PROCEDURE DefineInbuilts;

END Inbuilt.
```

INBUILT - IMPLEMENTATION

```
IMPLEMENTATION MODULE Inbuilt;

FROM DBase  IMPORT SymTabRec, ClauseRec,
                   SymTabPtr, ClausePtr,
                   SymType,   MKClauseRec;

FROM STable IMPORT Insert;


(* ------------------------------------------------------------------------ *)


        PROCEDURE DefineInbuilts;

           (* Make a symbol-table entry and one clause record for
              each inbuilt predicate.
           *)

           PROCEDURE Make( ID    : ARRAY OF CHAR;  (* name *)
                           Proc  : InBltProc;      (* procedure identifier *)
                           Arity : SHORTCARD);     (* arity *)

           VAR CPtr : ClausePtr;
               SPtr : SymTabPtr;
           BEGIN
               SPtr := Insert(ID,functor,Arity);
               CPtr := MKClauseRec();
               SPtr^.FstCls := CPtr;
               SPtr^.LstCls := CPtr;
               CPtr^.Next   := NIL;
               CPtr^.InBlt  := TRUE;
               CPtr^.Proc   := Proc;
               CPtr^.Entry  := SPtr;
           END Make;

        BEGIN
           Make ('var',isvar,1);
           Make ('!',cut,0);
           Make ('nl',nl,0);
           Make ('fail',fail,0);
           Make ('write',write,1);
           Make ('stats',stats,0);
           Make ('@<',lexlt,2);
           Make ('@>',lexgt,2);
           Make ('@>=',lexge,2);
           Make ('@=<',lexle,2);
        END DefineInbuilts;

END Inbuilt.
```

APPENDIX C
VRP Predefined Predicates - PREDEF.PRO

```
% --- predef.pro --- predefined prolog predicates for VRP ---
%     24.3.90


% --- true --- always succeeds.
      true.


% --- not (X) --- fails if X succeeds, succeeds if X fails. X must be
%                 instantiated or an error occurs.
      not (X) :- X,!,fail.
      not (X).


% --- nonvar(X) --- succeeds if argument is not a free variable.
      nonvar(X) :- var(X),!,fail.
      nonvar(X).


% --- = (A,B) --- succeeds if A and B are instantiated to the same term
                    instance. Otherwise fails.
      = (A,A).


% --- \=(A,B) --- not(=(A,B)).
      \= (A,A) :- !,fail.
      \= (A,B).


% --- call(X) --- executes X as a goal. X must be instantiated.
      call(X) :- X.


% --- repeat --- creates a bactrack point which succeeds infinitely.
      repeat.
      repeat :- repeat.


% --- a predicate (non-generative) version of member.
      memberp (H, [H|_]) :- !.
      memberp (X, [H|T]) :- memberp (X, T).


% --- the generative version of member.
      member (H, [H|_]).
      member (X, [H|T]) :- member (X, T).


% --- last element of a list.
      last (X,[X]).
      last (X,[_|Y]) :- last(X,Y).


% --- append second argument to first (list) giving third (list).
      append ([],L,L).
      append ([X|L1],L2,[X|L3]) :- append (L1,L2,L3).


% --- reverse first argument (list) giving second argument (list).
      reverse(L1,L2) :- rev2 (L1,[],L2).
      rev2 ([X|L],L2,L3) :- rev2 (L, [X|L2], L3).
      rev2 ([],L,L).
```

```
% --- return third argument (list) consisting of second argument (list)
%     without all occurences of first argument.
      delete (_,[],[]).
      delete (H,[H|T],X) :- !, delete (H,T,X).
      delete (X, [Y|T1],[Y|T2]) :- delete (X,T1,T2).


% --- subst (A1,L1,A2,L2) -- replace all occurences of A1 in list L1
                                        by A2 to give list L2
      subst (_,[],_,[]).
      subst (H1,[H1|T1],H2,[H2|T2]) :- !, subst (H1,T1,H2,T2).
      subst (X, [H|T1],Y,[H|T2]) :- subst (X,T1,Y,T2).


% --- write X followed by a CR/LF
      writeln(X) :- write(X), nl.


% --- write contents of a list as a string (ie without [] and separators)
      writestr([]).
      writestr([H|T]) :- write(H), writestr(T).


% --- premute(List1,List2) --- permute list1 returning result in list2.
      permute(L,[H|T]) :- append(V,[H|U],L),
                          append(V,U,W),
                          permute(W,T).
      permute([],[]).


% --- sort(List1, List2) --- sort the first list in ascending order,
%                            returning result in second. A quicksort
%                            is used (see [CLO81 p.157]).
      sort(L1,L2)                         :- qs(L1,L2,[]).
      split (H,[A|X],[A|Y],Z) :- @=<(A,H), split(H,X,Y,Z).
      split (H,[A|X],Y,[A|Z]) :- @>(A,H), split(H,X,Y,Z).
      split(_,[],[],[]).
      qs([H|T],S,X)                       :- split(H,T,A,B),
                                             qs(A,S,[H|Y]),
                                             qs(B,Y,X).
      qs([],X,X).
```