IMPLEMENTATION OF A LOGIC PROGRAMMING LANGUAGE: R-MAPLE
AND ITS EXTENSION TO INCLUDE FUNCTIONS

by

Benjamin Yu

B.Sc. (Hons.), The University of British Columbia, 1982.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1984

© Benjamin Yu, 1984
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 12 October, 1984

Abstract

The use of logic as a language for communication with computers has been exploited in various research areas. R-Maple is a logic programming language, designed by Paul Voda, which improves on some of the shortcomings of Prolog. The implementation details of R-Maple will be discussed. R-Maple is a relational programming language which has relations as its basic objects. It can be extended to include functions to form a functional and relational programming language. The advantage is to increase readability and efficiency during computation. RF-Maple is such a language which results from combining the features of R-Maple with a functional language F-Maple. The semantics of RF-Maple needs to be defined formally. The currently available methods of formalizing the semantics of a programming language are either too complex or insufficient for the task. The theory of pairs [Voda2 84] can be used in formalizing the semantics of RF-Maple. However, introduction of types to the theory is not straightforward. Hence RF-Maple is modified to become a typeless language and its semantics is defined using the theory of pairs.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>Logic Programming</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Implemented Systems</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Deficiencies of Existing Systems</td>
<td>6</td>
</tr>
<tr>
<td>3.</td>
<td>Introduction to R-Maple</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Concrete Syntax of R-Maple Programs</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Transformation as Computation</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Abstract Syntax of R-Maple</td>
<td>20</td>
</tr>
<tr>
<td>4.</td>
<td>Implementation of R-Maple</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Data Structure</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Construction of the R-Maple Tree</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Transformation of the R-Maple Tree</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>Processes</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Backward Movement of a Process</td>
<td>34</td>
</tr>
<tr>
<td>4.6</td>
<td>Free and Bound Variables</td>
<td>36</td>
</tr>
<tr>
<td>4.7</td>
<td>Hardware Consideration</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Examples of R-Maple Abstract Program</td>
<td>41</td>
</tr>
<tr>
<td>5.</td>
<td>Motivation for A Functional and Relational Programming Language</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Description of F-Maple</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Terms over F-Maple Types</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Examples of F-Maple Programs</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>Computation of F-Maple Programs</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>Description of RF-Maple</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>Examples of RF-Maple Programs</td>
<td>59</td>
</tr>
<tr>
<td>6.</td>
<td>Background of Semantics Theory</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>The Operational Approach</td>
<td>65</td>
</tr>
<tr>
<td>6.2</td>
<td>The Denotational Approach</td>
<td>66</td>
</tr>
<tr>
<td>6.3</td>
<td>The Axiomatic Approach</td>
<td>67</td>
</tr>
<tr>
<td>7.</td>
<td>Theory of Pairs</td>
<td>69</td>
</tr>
<tr>
<td>7.1</td>
<td>Semantics of RF-Maple</td>
<td>71</td>
</tr>
<tr>
<td>7.2</td>
<td>Typeless RF-Maple</td>
<td>72</td>
</tr>
<tr>
<td>7.3</td>
<td>Formalization of the Semantics of TRF-Maple in TP</td>
<td>76</td>
</tr>
<tr>
<td>8.</td>
<td>Future Work</td>
<td>88</td>
</tr>
<tr>
<td>9.</td>
<td>Conclusion</td>
<td>91</td>
</tr>
</tbody>
</table>
Acknowledgement

I would like to express my appreciation to Dr. Paul Voda for his continual support and guidance throughout the preparation of this thesis. His expertise and immense interest in programming languages and logic certainly motivates me to continue in my study and research. I would like to thank Dr. Karl Abrahamson for his careful reading of my thesis. I would like to thank the Natural Sciences and Engineering Research Council for their financial support. I would also like to thank my family, friends and my beloved fiancee for their continual prayer support and encouragement during the course of my program. Most of all, I would like to give all thanks and praises to my Lord and Saviour Jesus Christ, without Him this thesis would never have been completed.
Chapter 1

Introduction

1. Introduction

Recently, considerable amount of attention is given to the fifth generation computing especially with the Japanese effort in designing a whole new generation of computing machine. With the advance in machine technology, a parallel effort in designing software that utilizes the machine technology is necessary. Of foremost importance is the need of a programming language that will allow the user to interact with the computer more efficiently and with less mistakes being made. The programming language PROLOG which is used widely in many different areas of research in artificial intelligence, and before, PLANNER and CONNIVER, etc. all have received considerable attention. However, there are deficiencies in each of these languages. R-Maple grew out as a logic programming language that is based on predicate logic but with the added control feature that is lacking in other logic programming languages. Its features will be described in later sections. R-Maple is a logic programming language which does not allow functions as terms. This is a limitation as a logic programming language because this requires that all functions to be written as relations. Not only relations do not have the expressive power of functions, they require more overhead to compute due to their backtracking nature. To overcome this limitation, RF-Maple is designed to retain the features of a logic programming language like R-Maple but with the added feature of allowing functions as its terms. It is a combination of two languages, R-Maple and F-Maple. The latter being a typed language and
provides, not only for semantic extensibility (new types and functions), but also
for syntactic extensibility. In order to determine the correctness of the RF-Maple
language, it is necessary to formally define its semantics. The currently available
methods of formalizing the semantics are either too complex or inadequate for the
task. The theory of pairs, TP, is based on first order logic especially designed to
cater to formalizing the semantics of functional and logic programming language.
To define the semantics of RF-Maple it is necessary to introduce types into the
theory of pairs. This is not that straight forward. Hence, typeless RF-Maple is
designed to simulate the result that can be obtained in RF-Maple. The theory of
pairs is used to define the semantics of TRF-Maple.

This thesis is composed of three main parts. The first part is on the imple­
mentation of R-Maple. The second part is on the design of RF-Maple, and
finally, the third part is on the definition of typeless RF-Maple (TRF-Maple) and
the formalizing of its semantics using the theory of pairs. Chapter 2 will begin
with a background of currently available logic programming languages. Chapter
3 is the description of R-Maple and Chapter 4 is the implementation details of
the language. Chapter 5 is the description of RF-Maple, how the two languages
R-Maple and F-Maple are combined to form a functional and relational program­
ming language. Chapter 6 begins the background of the current methods in
defining the formal semantics of programming languages. Chapter 7 describes
the theory of pairs as a way of defining the semantics of a programming
language. Since introduction of types to TP is not that straight forward, typeless
RF-Maple (TRF-Maple) is designed that could simulate the computation of typed
RF-Maple. This section also shows how the semantics of TRF-Maple can be
formalized in the theory of pairs. Chapter 8 is a discussion on future work that may be continued.
Chapter 2

Logic Programming

2. Logic Programming

The idea of programming in logic can be traced back to the first logic programming languages, PROLOG, developed by Alain Colmerauer and Phillipe Roussel in 1972 [Kowa]. Logic is an important tool in the analysis and presentation of arguments. It investigates whether assumptions imply conclusions, independently of their truth or falsity and independently of their subject matter [Kowa 79]. Logic is used as a language for communication with computers because of its higher-level and more human-oriented nature than other formalisms specifically developed for computers [Kowa 79]. A programmer can initially ignore procedural details and concentrate on the (declarative) essentials of the algorithm. Having the program broken down into small independently meaningful units, it is much easier to understand. Design flaws are less likely to result, perhaps because it is more difficult to make a "logical error" in a program when its logic is actually expressed in logic [Warr 77].

The meaning of programs expressed in conventional languages is defined in terms of the behaviour they invoke within the computer. For example, a programmer using a conventional language, like Pascal, Ada, etc., needs to be aware of the incremental changes (or side effects) made to variables. On the other hand, the meaning of programs expressed in logic can be defined in machine-independent, human-oriented terms [Kowa 79]. There are no side effects in logic programs. The value of an expression (its meaning) is determined solely by the
values of its constituent parts. Thus should the same expression occurs twice in the same context, it denotes the same value at both occurrences.

2.1. Implemented Systems

The first logic programming system called PROLOG based on the procedural interpretation of Horn clause was designed and implemented in 1972. A Horn clause:

\[ B \leftarrow A_1, \ldots, A_m \quad m \geq 0 \]

is interpreted as a procedure whose body \( \{A_1, \ldots, A_m\} \) is a set of procedure calls \( A_i \). Top down derivations are computations. Generation of a new goal statement from an old one by matching the selected procedure call with the name \( B \) of a procedure is a procedure invocation [Kowa 79]. A more detailed description of PROLOG can be found in [Clos & Mell 80].

Another system which is also based on logic is the programming language PLANNER by Carl Hewitt [Hewi 69]. It was originally written in LISP. PLANNER accepts declarative information, as in predicate calculus. This is in the form of expressions which represent "facts" about the world. These are manipulated by the theorem-prover according to some fixed uniform process set by the system. PLANNER can also accept imperative information, telling it how to go about proving a subgoal, or to make use of an assertion. It has the full power of a general programming language to evaluate functions which can depend on both the data base and the subgoal tree, and to use its results to control the further proof by making assertions, deciding what theorems are to be used, and specifying a sequence of steps to be followed [Suss, Wino & Char].
Due to the inefficiencies of backtracking in PLANNER, another programming language also based on logic, called CONNIVER [Suss & Mcde 72] is developed. CONNIVER is a PLANNER-like programming language in which the programmer writes both problem-solving procedures and search strategies.

2.2. **Deficiencies of Existing Systems**

It has been demonstrated that PROLOG can be used in a variety of applications such as natural language systems [Colm 75], [Dahl & Samb 76], compiler writing [Colm 75] [Warrl 77], algebraic manipulation [Berg & Kano 75], definite clause translation grammar [Abra 84], and the automatic generation of plans and programs [Warr 74]. However, PROLOG does have its deficiencies. PROLOG has a very weak control component. Conceptually at least, the control component is entirely absent in PROLOG. The decision on how to sequence computations and what to do in parallel is left entirely to the PROLOG interpreter acting as a virtual executing machine [Voda 83]. Even concurrent PROLOG with the use of the commitment operator and read only variables is not that straightforward. The lack of control in PROLOG can be seen in the following example where the use of the cut operator is necessary.

\[
P(x) \leftarrow Q(x),!,R(x) \\
P(x) \leftarrow S(x) \\
\neg P(a)
\]

This is a very common situation when a programmer relies on implicit control built into the PROLOG interpreter. Assume that during the refutation of P(a) the test Q(a) is satisfied but not R(a). The next clause for P(x) should be tried. However, due to the use of the cut operator, no backtracking takes place. The
programmer might have intended the second clause to be as follows:

\[ P(x) \leftarrow \neg Q(x), S(x) \]

but because of the difficulties with negation in PROLOG clauses, and for reasons of efficiency, a cut operator is used. Cuts cannot be explained in logic within the context of a clause and without an ordering of clauses [Voda 83].

PLANNER and CONNIVER, on the other hand, allow control primitives for guiding search strategies. But this is exactly what is not desirable in a language where a programmer should only focus on the nature of the problem rather than the behaviour of the program inside the machine.
Chapter 3
Introduction to R-Maple

3. Introduction to R-Maple

R-Maple is a relational concurrent programming language based on predicate logic developed at the University of British Columbia by Paul Voda. The philosophy behind R-Maple is to allow the programmer to view the computation of programs as an attempt to prove the program using certain axioms and rules of inference. The advantage is that, in trying to solve a problem, the programmer can lay down statements that are true and let the computing machine draw conclusion from it, or to prove the theorem that he/she wants it to prove. This style of programming using logic is contrasted with the conventional way of programming using languages like Pascal, Ada, etc. in that, in the later case, a programmer needs to think also in terms of what the computer should do and when. [Hend 80] The following is a brief description of R-Maple. A full description of the language can be found in [Voda 83].

R-Maple is a relational programming language as opposed to a functional programming language. A functional programming language has functions as its basic objects and functions have a many-to-one correspondence between the arguments and the results (or between the domain and range). A relational programming language, on the other hand, has relations as its basic objects, and relations can have multiple results corresponding to an input argument. (ie. relations have a many-to-many correspondence between the domain and range). Since R-Maple is a relational programming language, this adds to R-Maple the possibility of
achieving non-determinism which is not available in functional programming language.

R-Maple is an applicative language as opposed to an imperative language. By applicative language, it means that the value for an expression or a relation (or its meaning) is determined solely by the values of its constituent parts. Thus, should the same expression occur twice in the same context, it denotes the same value at both occurrences. On the other hand, an imperative language computes by effect. This means that in using an imperative language, one must think in terms of the incremental changes made to variables by successive assignments [Hend 80]. The same procedure declared in an imperative language may produce different results due to the effectual assignments to variables. It is the absence of this ability to effect changes to values of variables which characterizes purely applicative languages like R-Maple.

As mentioned earlier, R-Maple is based on predicate logic. Programs of R-Maple correspond to formulas of predicate calculus. As will be seen in Chapter 7, the task of specifying the semantics of TRF-Maple where R-Maple is one of its component languages becomes rather straightforward.

The syntax of R-Maple can be viewed from two main aspects. The first aspect is called the concrete syntax which is a fairly rich syntactic structure that makes the program readable and easy to understand. The second aspect is called abstract syntax which is less readable by humans but may be easily composed and decomposed by other programs. The first part of this thesis which is on the implementation of R-Maple concentrates on this latter aspect, (ie. taking a pro-
gram written in abstract syntax and interpreting it). A programmer writing a
R-Maple program would, however, use the concrete syntax. Thereafter, it is con­
verted into the abstract syntax by a compiler, and finally executed by the R-
Maple interpreter.

The concrete syntax of R-Maple will be described first followed by the abstract
syntax.

3.1. **Concrete Syntax of R-Maple Programs**

R-Maple is a logic programming language, where, each program of R-Maple
can be read as a formula of predicate logic. A formula written in first order
predicate logic can be easily translated into the concrete syntax of R-Maple
because there is a correspondence between keywords used in R-Maple with those
used in logic. The following table summarizes the correspondence.

<table>
<thead>
<tr>
<th>R-Maple:</th>
<th>Logic:</th>
<th>R-Maple:</th>
<th>Logic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>formula</td>
<td>expression</td>
<td>term</td>
</tr>
<tr>
<td>prog. variable</td>
<td>indiv. variable</td>
<td>invocation</td>
<td>atomic formula</td>
</tr>
<tr>
<td>find</td>
<td>$\exists$ quantifier</td>
<td>; and $|$</td>
<td>conjunction</td>
</tr>
<tr>
<td>or and orp</td>
<td>disjunction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hence, an R-Maple program such as:

\[
\text{find } x \text{ in } x := 5; x > 3
\]
can be directly translated to a first order logic formula as:

\[
\exists x (x = 5, & x > 3).
\]

Also, the program written in R-Maple as:

\[
\text{find } x \text{ in } (x := 5 \text{ or } x := 6 \text{ or } x := 77); x > 10.
\]
can be written as a first order logic formula as:
R-Maple provides sequential and parallel connectives. That is, if $A$ and $B$ denote two programs, then $A ; B$ denote the logic formula $A \& B$. Moreover, it tells the R-Maple interpreter that $A$ is to be executed and completed before $B$ is to be executed. This is contrasted with the parallel connective $||$ where $A || B$ instructs the interpreter to start a process in both $A$ and $B$ in parallel. The meaning of $A ; B$ and $A || B$ are the same since both denote the logic formula $A \& B$. But the control, which directs the behaviour of the machine is different. This might have significant consequences in proving some theorems on the R-Maple machine such as:

$$p(x); (5 > 7)$$

where computation of $p(x)$ diverges. The corresponding formula in first order logic is false since 7 is always greater than 5. However, to compute this formula in R-Maple using the sequential connective will lead to an infinite loop or a deadlock since the predicate $p(x)$ diverges. On the other hand, computing $p(x) || (5 > 7)$ on the R-Maple machine will correctly produce the answer. It should be noted that proving the following theorem on the R-Maple machine produces the correct result even though $p(x)$ diverges.

$$false ; p(x)$$

Computation will converge with the correct result $false$.

Sequential and parallel disjunctive are also provided in R-Maple and their behaviour in the executing machine is similar to the case described for the connectives earlier.
Conditional tests are also provided in the concrete syntax of R-Maple. They take the usual form of the IF-THEN-ELSE statement. For example,

\[
\text{if } x > 3 \text{ then } s(x) \text{ else } t(x).
\]

Its declarative reading is:

\[
(x > 3) \rightarrow s(x) \quad \& \quad (x \leq 3) \rightarrow t(x).
\]

A user may use predicates predefined by the system or those defined by the user himself or herself. Each predicate can take the form with output arguments or without. Those that take the first form are called tests and those that take the second form are called generators. A bar | is used to separate the input arguments from the output arguments in the concrete syntax of generators. Some examples of system predefined tests are: eq(x,y), ne(x,y), lt(x,y), etc., which are tests for equality, inequality, and other comparison tests for two numbers x and y. Examples of system predefined generators are: add(x,y |z), sub(x,y |z), div(x,y |z), rem(x,y |z), etc., which are generators for addition, subtraction, division, and the remainder function etc. for two numbers x and y with the output in z. The user is given the freedom to use either infix or prefix form for the system defined predicates. In infix form, the user is allowed to use symbols such as +, -, >, etc. instead of the function name. All infix forms are converted to prefix forms during the compilation process from concrete to abstract syntax. For the rest of the discussion on R-Maple, all system defined predicates used in the examples are presented in infix form.

Users can define his/her predicates by preceding the program with a predi-
cate name and the input and output variables (if any). Thus, a predicate to test whether a number is even or not can be first defined in first order logic as:

\[
\text{Even}(x) = \exists r \ (\text{rem}(x,2,r) \land r = 0)
\]

where \( \text{rem}(x,y,z) \) is a predicate to compute the remainder of \( x \) divided by \( y \) with the result in \( z \). Since there is a predefined generator "rem" and also the predefined test "eq" (for test of equality) in R-Maple, the formula can be simply translated as:

\[
\text{Even}(x) \quad \text{is find} \ r \ \text{in} \ \text{rem}(x,2 \mid r); \ \text{eq}(r,0).
\]

The user can then use the predefined predicate in other parts of another program. R-Maple does not allow the output variable of a predicate to be initially instantiated during the invocation. Hence, although the declarative reading for the \( \text{Even} \) predicate is also:

\[
\text{Even}(x) = \text{rem}(x,2,0)
\]

R-Maple does not allow the predicate \( \text{rem} \) to be invoked as \( \text{rem}(x,2 \mid 0) \). On the other hand, the input argument may be left uninstantiated during the invocation. The use of the uninstantiated input argument will be suspended until another process assigns a value to it. This will be discussed in more details in later sections.

Although R-Maple allows the definition of predicates with any number of input arguments, in essence, the compiler translates the input and output arguments to a single argument as an expression. The syntax of an expression is as follows.
The simplest form of an R-Maple expression can be a literal or a variable.

The formation rules for a literal are:

a) each numeral is a literal,

b) the symbol "nil" is a literal,

c) if s and t are literals, so is [s,t].

If $Expr_1$ and $Expr_2$ are expressions, then so is $[Expr_1,Expr_2]$. This final form is often referred as a consed expression and resembles closely to that in LISP.

Appendix A gives the full description for R-Maple expressions in BNF notation.

The input and output arguments of a predicate is translated to an expression of the form $[Expr_1,y]$ where y is the output argument and $Expr_1$ is the expression for the input arguments. The latter can be either a single argument or any number of arguments represented as a single expression. For example, the invocation of a predicate $foo$ with three input arguments, w, x and y, and one output argument, z, is taken to be a predicate with an argument of the form $foo ([[w,[x,y]],z])$. To allow accessing of the individual argument, R-Maple provides the find construct to be of the form:

$$\text{find} \ [hd,tl] = a \ \text{in} \ Prog$$

where $a$ can be “split” into two expressions with $hd =$ head of the pair $a$ and $tl =$ tail of the pair $a$ for further processing. This construct can also be referred as a split operation in R-Maple. Hence, for example:

$$foo (x) =$$
$$\text{find} \ [hd,tl] = x \ \text{in} \ \text{find} \ [hd\ 1,tl\ 1] = hd \ \text{in}$$
$$\text{print} (hd\ 1); \ \text{print} (tl\ 1); \ tl := hd\ 1.$$  

This predicate can be invoked in a program as:

$$\text{find} \ y \ \text{in} \ foo ([[1,2],y]).$$
When executed, the program will print the value 1 which is the head of the head of the input argument, and also the value 2 which is the tail of the head of the input argument. Finally, the predicate is satisfied with \( y = 1 \).

Appendix B gives the full description for R-Maple programs and the definition of predicates in BNF notation.

### 3.2. Transformation as Computation

Computation of a R-Maple program is a series of transformation until the most simple atomic formula \(<\text{trueop},\text{nil}>\) or \(<\text{falseop},\text{nil}>\) results. A computation proceeds by steps which successfully modify the original program. The following describes briefly the transformation schema that are applicable for the R-Maple machine. The transformation schemas are listed out in Appendix C in a more formal and rigorous format. They appear also in [Voda 83].

Computation is performed by application of rewriting rules of the form \( A \Rightarrow B \) where both sides contain the operator \(!\).

Transformation of system defined predicates \( \text{Lt} \), \( \text{Le} \), \( \text{Gt} \), and \( \text{Ge} \) to true or false can proceed only if the arguments are not variables. Otherwise, transformation is suspended until the appropriate arguments are obtained.

Predicates of identity, \( \text{Eq} \) and \( \text{Ne} \) do not require both arguments to be fully reduced. This sometimes permits an earlier decision on expression still containing variables. When two expressions of the form \([Expr_1,Expr_2]\) and \([Expr_3,Expr_4]\) are tested for equality, two processes are created to compare \( Expr_1 \) with \( Expr_3 \) and \( Expr_2 \) with \( Expr_4 \) in parallel. More sub-processes may be created from these
processes. If any of these processes reduces to false even if some of the other
processes are not finished in their execution, the entire test can be quickly
reduced to false.

Invocation of the predicate Print is delayed until its arguments becomes a
quoted expression. Then the literal is printed out and the invocation is reduced
to T!

Arithmetic predicates, Add, Sub, Mul, Div, and Rem, are also delayed until
the appropriate arguments are obtained. In the case of user defined predicates,
invocation of predicates is first preceded by substitution of the formal argument
with the actual argument. The process may then proceed to transform the body
of the invoked predicate.

Invocation of the predicate Return is a bit different from the invocation of
other predicates. This will be discussed later.

Other unconditional schemas of transformations include the negation
scheduling, conjunction scheduling, disjunction scheduling, truth table transfor-
mations. These transformations preserve the intended meaning unconditionally,
i.e. they do not rely on lists of definitions. Most of these transformations are
relatively straight forward. One thing to note is that in the transformation of a
parallel disjunction or conjunction, two processes are created to execute the left
and right branch in parallel whereas in the sequential case, the original process is
used to execute the left branch. Only when the transformation of the left branch
is reduced to true or false will the right branch be transformed.
Decisions can be expressed with the help of sequential or parallel and's, or's, and not's but the efficiency of computations is improved by having if-then-else as a basic operator. In the transformation of

```plaintext
if Prog_1 then Prog_2 else Prog_3,
```

once Prog_1 is reduced to true or false, the if-then-else construct is transformed into either Prog_2 or Prog_3. Execution continues with their transformation.

The execution of the split operation (or

```plaintext
find [ Var_1, Var_2 ] = Expr in Prog
```

) is delayed until the expression to be split is not a program variable. A consed expression can be immediately split into two constituents. Constituents of the consed expression are then substituted in Prog for the appropriate variable.

Transformation for the `find` construct is a search operation which introduces a scope in which a value or values of the variable declared will be searched for. Execution immediately transfers to the body of the find construct to search for the value of the variable.

When the value for a variable is "found", the predicate Return is invoked so that the find construct for that variable may be discharged. For every invocation of the predicate Return for a variable "x", there needs to be a corresponding `find` for the variable "x". In R-Maple, when the predicate Return is invoked, the value that is found for the variable should be "moved" back to the `find` so that the searching process may be discharged. [Note that the schema for the transformation of the find construct is a forward movement of the process, allowing the body to be executed. The Return predicate serves as an initiation of the back-
ward movement of the process to discharge the \texttt{find}. During the backward movement of the variable to its corresponding \texttt{find}, there might be other \texttt{find} along the backward move. For example:

\[ \texttt{find } x \texttt{ in } ... \texttt{find } y \texttt{ in } ... x := 4 ; ... ; y := 6 ... . \]

or represented in logic as:

\[ \exists x \ldots \exists y (... x = 4 ... y = 6 ...) \]

Also, during the backward move, a parallel or sequential "and" or "or" may be encountered. In those cases, there needs to be some mechanisms to store the type of nodes that are encountered along the backward move so that further execution may be resumed later. The transformation schema for the predicate Return is:

\[ \text{Return (Expr | Var ) } \Rightarrow \texttt{bfindss } \alpha \texttt{ in } (\texttt{Var := Expr ; T}) \texttt{ or F} \]

where \( T \) represent true and \( F \) represent false. \texttt{bfindss} is one of four new constructs to be introduced into the transformation schemas of R-Maple. The other three are: \texttt{bfindsp}, \texttt{bfindps}, \texttt{bfindpp}. The prefix of these "opcodes", \texttt{bfind}, indicates that these are for backward find. All have the syntax of the form:

\[ \texttt{bfind } \alpha \texttt{ in } (\texttt{Var := Expr and Prog}_1 \texttt{ or Prog}_2) \texttt{ or Prog}_2 \]

\( \texttt{Prog}_1 \) and \( \texttt{Prog}_2 \) are R-Maple programs and \texttt{bfind} must be one of \texttt{bfindss}, \texttt{bfindsp}, \texttt{bfindps}, or \texttt{bfindpp}. The postfix "ss", "sp", "ps", and "pp" are used when the find is finally discharged. They are used to indicate whether the \texttt{and} and \texttt{or} are sequential or parallel. "ss" indicates that both \texttt{and} and \texttt{or} are sequential operations, "sp" indicates that \texttt{and} is sequential and \texttt{or} is parallel, while "ps" indicates that \texttt{and} is parallel and \texttt{or} is sequential, and "pp" indicates that both \texttt{and} and \texttt{or} are parallel operations. Their use is apparent in the
transformation schemas presented in Appendix C. These transformation schemas indicate how the postfix of \texttt{bfind} is changed so that information about the sequential or parallel boolean operations on the move backwards may be preserved. Also, in some cases, a new process may be started to preserve the meaning of parallel boolean operations.

So far, nothing has been said about \( \alpha \). \( \alpha \) is a list of variables. Initially, when \texttt{Return} is invoked, \( \alpha \) is empty. During the backward move, the value of a variable \( x \) may be moved back through a \texttt{find} on a different variable \( y \). In this case, \( y \) will be added to the list of \( \alpha \).

The rewriting continues until the program is transformed into the form where no rewriting rules are applicable. This can either fail to terminate, terminate normally (in the form \( T! \)), or remain deadlocked. It should be noted that the executing machine is not a full theorem prover and that if the program never terminates, it does not mean that the original program was not a theorem. (For instance, the earlier example : \( P; 5>7 \) will never terminate if \( P \) does not terminate although the declarative reading of the formula is false. But since the sequential conjunction ; is used, the executing machine will try to compute \( P \) before starting to compute \( 5>7 \) and therefore the whole program will never terminate.)

It should be noted that computations of R-Maple are \textit{invariant} to the declarative reading of programs. This is because each rewriting rule is justified by a logical tautology. For example, computations for tests employ the truth tables of logical and the moving back of a value for a variable to its corresponding
"find" is possible because of the tautologies:

\[
\begin{align*}
\exists y \ (x = a & \ A \ V \ B ) & \ & C \ \leftrightarrow \\
\exists y \ \{x = a & \ (A & \ C ) \ V \ (B & \ C )) \}
\end{align*}
\]

\[
\begin{align*}
\exists y \ (x = a & \ A \ V \ B ) \ V \ C \ \leftrightarrow \\
\exists y \ \{x = a & \ A \ V \ (B \ V \ C )) \}
\end{align*}
\]

provided \( y \) does not occur in \( C \). If \( y \) occurs in \( C \), it must be systematically changed to a different variable.

There are no rewriting rules for guiding an assignment through a negation. This is because there is no good declarative reading for such a transformation. A program that attempts this will result in a deadlock. Moreover, there is no need for this in logic programs as the practice of Prolog confirms.

By the employment of logical connectives, the use of explicit quantifiers (\( \textit{find} \)) coupled with the use of \( \textit{if} \) statements, all the cuts and commits of Prologs can be eliminated. Moreover, a wide scale of control behaviours is possible without compromising the declarative reading of programs [Voda & Yu 84].

3.3. **Abstract Syntax of R-Maple**

Programs written in concrete syntax are compiled into abstract syntax. Abstract R-Maple programs are directly interpreted on the R-Maple machine. The following is a summary of the formation rules for abstract R-Maple programs. They are slightly different from that described originally in [Voda 83] and this syntax is recognized by the currently implemented R-Maple interpreter. Meta-variables which are syntactic variables ranging over terms, are used in the description of the formation rules. The following syntactic variables will be used.
Syntactic variables are used in the standard way of logic and they should not be confused with non-terminals of context-free grammars. Syntactic variables of the same kind, but with different subscripts, range over their domains independently. Two or more occurrences of the same syntactic variable in a schema have to be replaced by the same sequence of symbols.

Formation rule for program variables:

\[ \langle \text{varop}, \text{Num} \rangle \] is a program variable.

Formation rule for program predicates:

\[ \langle \text{predop}, \text{Num} \rangle \] is a program predicate.

Formation rules for expressions:

a) \[ \langle \text{quoteop}, \text{Lit} \rangle \] and \[ \text{Var} \], are expressions.

b) If \[ \text{Expr}_1 \] and \[ \text{Expr}_2 \] are not both quoted then \[ \langle \text{pairop}, \text{Expr}_1, \text{Expr}_2 \rangle \] is an expression.

An expression is quoted if it is of the form \[ \langle \text{quoteop}, \text{Lit} \rangle \]. An expression is consed if it is of the form \[ \langle \text{pairop}, \text{Expr}_1, \text{Expr}_2 \rangle \].

Formation rules for programs:

\[ \langle \text{trueop}, \text{nil} \rangle, \quad \langle \text{falseop}, \text{nil} \rangle, \quad \langle \text{attrop}, \text{predop}, \text{Num}, \text{Expr} \rangle, \]
\[ \langle \text{ifop}, \text{Prog}_1, \text{Prog}_2, \text{Prog}_3 \rangle, \quad \langle \text{findop}, \text{varop}, \text{Num}, \text{Prog} \rangle, \]
\[ \langle \text{splitop}, \text{varop}, \text{Num}_1, \text{varop}, \text{Num}_2, \text{Expr}, \text{Prog} \rangle, \]
\[ \langle \text{orsop}, \text{Prog}_1, \text{Prog}_2 \rangle, \quad \langle \text{orpop}, \text{Prog}_1, \text{Prog}_2 \rangle, \]
\[ \langle \text{andsop}, \text{Prog}_1, \text{Prog}_2 \rangle, \quad \langle \text{andpop}, \text{Prog}_1, \text{Prog}_2 \rangle, \quad \langle \text{notop}, \text{Prog} \rangle, \]
\[ \langle \text{defop}, \text{predop}, \text{Num}_1, \text{varop}, \text{Num}_2, \text{Prog} \rangle \] are programs.

The formation rules for the last group of operators in R-Maple are:
This last group of operators are called processes or program-counters. They indicate where in the program transformation takes place. These operators are constructed by the executing machine and are not used in the translation from concrete syntax to abstract syntax. Their use is equivalent to the \texttt{bfind} operator described in the transformation schema in the last section. More will be discussed on this latter. The symbol \texttt{!} is used to denote a process in execution. Thus, \texttt{P!} indicates that there is a process executing the program \texttt{P}. Since \texttt{wheress}, \texttt{wheresp}, \texttt{whereps}, and \texttt{wherepp} denote a processes in execution, the symbol \texttt{!} is included in their formation rules. Each formation rule is delimited by the angle brackets (although in the actual input to the R-Maple interpreter, they are not used). The keyword that follows each open angle bracket \texttt{<} can be thought of as an opcode for the R-Maple machine and the rest being the arguments for the opcode. Hence, in the following discussion, the terms “keyword” and “opcode” will be used interchangeably.

As can be seen in the formation rules for the abstract syntax of R-Maple, it is closely related to the concrete syntax. For each variable used in a R-Maple program, it is translated to the abstract form of \texttt{<varop,Num>}. Two or more occurrences of the same variable in the same context are translated with the same value for \texttt{Num}. Similarly, program predicates are translated to the abstract form of \texttt{<predop,Num>}.

The abstract syntax for a literal is \texttt{<quoteop,Lit>}. Thus, the numeral
672 is represented in the system as \(<quoteop,672>\). Since a literal is an expression, \(<quoteop,\text{Lit}>\) is said to be a quoted expression. Given two expressions \(\text{Expr}_1, \text{Expr}_2\) which are not both quoted, then \(<\text{pairop,Expr}_1,\text{Expr}_2>\) is an expression. This concept is slightly modified in the following discussion and in the implementation of R-Maple. Pairop is used also in pairing two quoted literals as well as non-quoted literals. That is, no matter what \(\text{Expr}_1\) and \(\text{Expr}_2\) stand for, \(<\text{pairop,Expr}_1,\text{Expr}_2>\) is also an expression. This corresponds to \([\text{Expr}_1,\text{Expr}_2]\) in the concrete syntax. An expression of the form \(<\text{pairop,Expr}_1,\text{Expr}_2>\) is referred to as a consed expression in R-Maple abstract syntax. The concept of consed and quoted expressions corresponds to the consed and quoted S-expressions in LISP. Note that the modified pairop construct resembles closely to the cosed expression in LISP. A list is represented in R-Maple using the \text{pairop} operation. All lists are represented as pairs of sub-lists. A list must be terminated by the literal nil at the end. This is slightly modified in the implementation since a 0 is used instead to indicate end of a list. For example, the list \([1,2,3,4,0]\) is taken as \([1,[2,[3,[4,0]]]]\) and is represented in abstract R-Maple syntax as :

\(<\text{pairop,quoteop,1}>,<\text{pairop,quoteop,2}>,<\text{pairop,quoteop,3}>,<\text{pairop,quoteop,4}>,<\text{quoteop,0}>>>>>

Since R-Maple is a logic programming language, every formula to be proved is to be reduced to either true or false. Therefore, the system provides the programs \(<\text{trueop,nil}>,\) and \(<\text{falseop,nil}\. \text{attrop,predop,Num,Expr}>\) is translated from a call to a predicate where \text{Num} is the unique value assigned to the predicate when it was defined and \text{Expr} is the argument to the predicate.
**Expr** may be a consed or a quoted expression. The former allows more than one input arguments to be passed into the predicate program. For example, if the predicate expects two input arguments and one output arguments, then passing the actual input arguments of the numbers 53 and 63 and the output argument of \(<varop,33>\) calls for an expression of the form:

\(<\texttt{pairop},\texttt{pairop},\texttt{quoteop 53},\texttt{quoteop 63},\texttt{varop},33>\).

The if-then-else concrete syntax of R-Maple is translated to 
\(<\texttt{if op},\texttt{Prog}_1,\texttt{Prog}_2,\texttt{Prog}_3>\) where \(\texttt{Prog}_1\) is a test to be reduced to true or false. If true, \(\texttt{Prog}_2\) will be executed, else \(\texttt{Prog}_3\) will be executed.

The find statement is translated to 
\(<\texttt{findop},\texttt{varop},\texttt{Num},\texttt{Prog}>\) where \texttt{Num} is the number assigned to the output variable and which is to be “found” in the program \(\texttt{Prog}\). As can be seen, this is a direct translation of the find statement in concrete syntax to the abstract syntax. The other form for \texttt{find}\ where an expression can be split into two of its constituents such as:

\texttt{find [hd | tl] = a in Prog}

has its corresponding abstract syntax as:

\(<\texttt{splitop},\texttt{varop},\texttt{Var}_1,\texttt{varop},\texttt{Var}_2,\texttt{Expr},\texttt{Prog}>\)

where \(\texttt{Expr}\) must be a consed expression. \(\texttt{Expr}\) will be split into two expressions. The head will be bound to \(\texttt{Var}_1\) and the tail will be bound to \(\texttt{Var}_2\). These variables can then be used in the evaluation of the program \(\texttt{Prog}\).

The sequential and parallel conjunctive and disjunctive constructs of R-Maple are translated to 
\(<\texttt{andsop},\texttt{Prog}_1,\texttt{Prog}_2>,\texttt{andsop},\texttt{Prog}_1,\texttt{Prog}_2>,\texttt{orsop},\texttt{Prog}_1,\texttt{Prog}_2>,\texttt{orsop},\texttt{Prog}_1,\texttt{Prog}_2>\) where \texttt{andsop} indicates a
sequential and, \textit{andpop} indicates a parallel and, \textit{orsop} indicates a sequential or, and finally, \textit{orpop} indicates a parallel or. \texttt{Prog}_1 \text{ and } \texttt{Prog}_2 \text{ are to be reduced to true and false (if possible) to allow the boolean comparison to proceed. The last boolean operation is the negation operation. Its abstract syntax is of the form } <\textit{notop},\texttt{Prog}_1>.

User defined predicates in R-Maple is translated into abstract syntax of the form <\textit{def op},\textit{predop},\texttt{Num}_1,\texttt{varop},\texttt{Num}_2,\texttt{Prog}>. \texttt{Num}_1 \text{ is assigned by the translator from concrete syntax to abstract syntax to give the predicate a unique number so that the predicate body can be invoked later. } \texttt{Num}_2 \text{ is the number assigned to the variable used as the argument in the program body } \texttt{Prog}. \text{ As seen in the presentation of the concrete syntax of R-Maple, this does not restrict the user to define predicates using only one argument since multiple arguments may be retrieved from the formal argument variable by the "split" operation in the body of the program } \texttt{Prog}. \text{ For example:}

\texttt{<def op,predop,45,varop,3,<splitop,varop,1,varop,2,varop,3,....,>>}

This predicate can then be invoked, say, with the arguments:

\texttt{<pairop,<quoteop,123>,<quoteop,432>>}.

The rest of the formation rules for the abstract syntax of R-Maple including \textit{wheres}, \textit{wheresp}, \textit{whereps}, and \textit{wherepp} are used for returning values assigned to variables during the execution of the program. They are not used in the translation of R-Maple programs from concrete syntax to abstract syntax. All have the syntax of the form:

\texttt{< where, a,Var,Expr,Prog}_1,\texttt{Prog}_2>
where must be one of \(\text{wheress}, \text{wheresp}, \text{whereps},\) or \(\text{wherepp}\). Since the user never generates code using the formation rules for processes, there is no direct corresponding concrete syntax to the abstract syntax of \(\text{where}\). However, in the discussion for backward move of a value to its corresponding \(\text{find}\), the \(\text{bfind}\) operator was introduced in the transformation schema. This is, in essence, the corresponding concrete syntax for the abstract syntax of \(\text{where}\). The terms \(\text{bfind}\) and \(\text{where}\) will be used interchangeably for the rest of the discussion.

Usually, when the term \(\text{bfind}\) is used, the transformation schemas are being referred to, whereas when the term \(\text{where}\) is used, the abstract syntax is being referred to.

Appendix D summarizes the formation rules of the abstract syntax discussed so far by showing the equivalence between the abstract syntax and their semantics in first order logic. The meaning component of an abstract program will be given by a naming function denoted by "\(*\)" which is defined on the domain of abstract program. Thus the intended meaning of a program \(A\) is the meaning of the formula \(A^*\).
Chapter 4

Implementation of R-Maple

4. Implementation of R-Maple

R-Maple was written in C and runs under the Unix† operating system Version 7. The basic operations of the R-Maple machine are 1) accept input in abstract syntax, 2) build a binary tree structure based on the input, 3) manipulate the tree by transforming it according to the transformation rules discussed in Section 3.2. Each of these operations will be described in more details in the following sections. First, the data structures used in the implementation will be discussed.

4.1. Data Structure

The binary tree to be constructed is stored in an array [1, max_storage_size] of records where max_storage_size is currently 2000. Each record, containing 7 fields, is used to represent a node. Since the tree to be constructed is a binary tree, there are two main fields in each record which store the data value for the left and right branch of a node. Their field names are head_value and tail_value. A node may contain the address of another node or a data value. Therefore, a bit is used to indicate for each of these two fields, head_value and tail_value, whether the content is a pointer or a data value. These are the fields is_ptr_in_head and ptr_is_in_tail. They are set to 1 if the head_value or the tail_value is a pointer, and 0 if it is a data value which can either be an opcode,

† Unix is a trademark of Bell Laboratories.
Section 4.1  Data Structure

A numeral, etc., or a data value. Each node has also a field to indicate which process is currently executing the node. The field is referred to as ex_proc. Processes will be discussed next. A backward pointer, backward_ptr, is also included in the record for pointing at a parent node in the tree. Its use will become apparent later. The structure of each node is represented in C as:

```c
struct cell
{
  unsigned is_ptr_in_head:1;
  unsigned ptr_is_in_tail:1;
  int    ex_proc;
  int    head_value;
  int    tail_value;
  int    backward_ptr;
}
```

Appendix E shows the basic tree structures that will be constructed corresponding to each formation rules of the abstract syntax of R-Maple. Only the fields head_value and tail_value are shown.

Once the tree is constructed, a process is assigned to transform the tree starting at the root until it is reduced to true or false. As will be seen later, execution of the abstract program is equivalent to the transformation of the tree structure. Therefore, the terms "execution" and "transformation" will be used interchangeably for the rest of this discussion. A process may be required to traverse down the tree to reduce a subtree before further transformation of the entire tree may proceed. Information about which part of the tree a process is executing and its status are stored in a process control block (PCB). Each process has its own PCB. The data structure for the process contains also 7 fields. These include 4 boolean flags which indicate whether the process is unassigned,
running, suspended, or finished. The use of these flags will be explained in the section on Processes (Section 4.4). The data structure for a process control block is encoded in C as follows.

```c
struct process_control_block
{
    unsigned unassigned : 1;
    unsigned running : 1;
    unsigned suspended : 1;
    unsigned finished : 1;
    int block_process;
    int proc_root;
    int prog_mark;
}
```

### 4.2. Construction of the R-Maple Tree

The input function of the interpreter has two main subfunctions: 1) to ensure that the abstract syntax is correct, and 2) to build the tree structure. A one-symbol-look-ahead parser is used to analyse the input. At the top entry level to the parser, a program or a definition of a predicate is expected. A program can be recognized by the first keyword which must be one of `trueop`, `falseop`, `attrop`, `andsop`, `andpop`, `orsop`, `orpop`, `notop`, `ifop`, `findop`, and `splitop`. A definition can be recognized by the keyword `defop`. Once the keyword is recognized, the parser can recursively call itself to parse the rest of the input whenever the parser expects a sub-program, (eg. `<ifop, Prog1, Prog2, Prog3>`), or it can call another procedure to parse a variable or an expression etc.. The abstract program can span several input lines and must be terminated by a period.

The binary tree is built through a call to a procedure to allocate a cell and then filling the cell with appropriate information. In many cases, the head value
contains the data and the tail value contains a pointer. For example, 
<quoteop,3> is represented as:

```
quoteop
   |
---|--
   |
  3
```

A nil pointer is represented by setting the bit is_ptr_in_head or ptr_is_in_tail (depending on whether it is the head or the tail that contains the nil pointer) to 1. The contents for head_value or the tail_value will then contain the value -1. In some cases when both the head value and tail value are pointers to other nodes, the two bits is_ptr_in_head and ptr_is_in_tail are both set to 1.

4.3. **Transformation of the R-Maple Tree**

Once the tree is constructed, a process is assigned to the root of the tree. The field proc_root and prog_mark are updated with the address of the root node. Execution of a R-Maple tree is equivalent to the transformation of the tree structure based on the transformation schema described in section 3.2. Hence, computation of R-Maple programs reduces to a sequence of pruning the tree, modifying the structure, copying parts of the tree etc. As will be seen clearer in Chapter 7, the transformations of a TRF-Maple tree which is similar in concept as the transformations of a R-Maple tree is not only a syntactic manipulation but they are also semantically correct.
4.4. Processes

A process can be in any one of four states. It can be either unassigned, running, suspended, or finished. A process that is unassigned is an idle process. It is ready to be assigned for the execution of any parts of a tree structure. Initially, all processes are unassigned. A process is running when it is already assigned a subtree to be executed and there is a processor assigned to the process. The corresponding PCB field is set to 1 when the process is selected to start the execution by a processor. It is reset to 0 when the process is blocked and another process starts its execution. Each process is allowed to continue to execute until one of the following conditions is encountered. In this case, the process is said to be suspended (or blocked). This allows a fair distribution of the use of the processors to each process. A process is suspended when 1) the process needs to traverse to one of the child nodes to execute a subtree. In this case, the pointer to the current tree node (prog_mark) is changed to point to the child node. The backward pointer of the child node is updated to point to the current node. The process is then stopped temporarily for other processes to proceed. 2) A process can be stopped when the current process is explicitly blocked by another process. An example is in the execution of andpop or orpop. The process executing at either of these nodes will be blocked while two new processes will be assigned to execute the left and right subtree in parallel. Execution of the parent process will resume when one of the child processes is finished. 3) When there is insufficient data for further processing, the process will also be stopped. This may happen during a comparison test such as $x > 5$ where $x$ is still a variable. Such situation occurs when there is a parallel process to assign $x$ a value. In this case, the
process executing $x > 5$ will not be able to proceed until the assignment is made. In this case, no change is made to the PCB for this process. A process is suspended (or blocked) when at least one other process must execute to completion before the process may assume execution. Finally, a process is finished when it has succeeded to reduce a subtree to either true or false.

The node or the root of the subtree where each process is originally assigned to execute is stored in the field proc_root of the PCB. This field is used to terminate the process later if the finished flag is set and the process is currently pointing at the root. Another field prog_mark is used to store the node in the tree structure where the process is currently manipulating. The value in prog_mark is changed whenever the process needs to traverse up or down the tree. This field is checked to see if it is equal to proc_root when the finished flag is set. If they are equal, this process will be terminated as mentioned above. The last field block_process is used to store the process number that is blocked by the current process. The blocked (or suspended) process will be reactivated when the current process is finished and terminated.

When a process reduces a subtree to either true or false, or when the predicate Return is invoked, it begins an upward traversal back towards the root of the tree where it was originally being activated. This is achieved by tracing the pointer stored in the field backward_ptr. Further transformations and backward move continues until the process reaches its root whereupon it is terminated and become unassigned. There are two cases when a backward move of a process is invoked. One is when a process successfully transformed a subtree to true or
false. The second case is when the predicate \textit{Return} is invoked. In both cases, a backward move of the process will begin.

Processes are assigned as specified in the transformation schema. There is a master process which continually monitors the status of the processes. When a process terminates execution, the rest of the processes are scanned to see if any of them may resume execution. Execution may result in a deadlock when all processes are not able to transform their subtrees. An example is:

\[
\text{find } x \text{ in } x > 5 \parallel \text{print } (x).
\]

In this case, the processes are waiting for the assignment of a value to the variable \( x \). The result is that each process will keep on trying to transform the tree but with no avail. Processes can be killed by other processes. Hence, even if a process goes into deadlock forever, another process may kill the process thus allowing the whole program to finish. For example:

\[
\begin{align*}
\text{find } x \text{ in } \text{print } (x) & \text{ orp } T \Rightarrow \\
\text{find } x \text{ in } \text{print } (x) & \text{! orp } T ! \Rightarrow \\
\text{find } x \text{ in } T ! & \Rightarrow \\
&T !.
\end{align*}
\]

Also, processes are killed or terminated when subtrees are deallocated. In the above example, if \( x \) is assigned a value eventually, the subtree for \text{print } (x) is deallocated because of the transformation schema:

\[
\text{Prog } \text{! orp } T ! \Rightarrow T !
\]

Hence, the process executing \text{print } (x) is terminated. When copying subtrees, the processes in the subtree are also copied. This means that new processes are created for the new subtree. For example, during the backward move of the
where node to an andpop node, both the subtrees branching from the andpop node contains processes. Therefore, during the transformation, the branch that is copied must also have new processes created for the copied structure. This is achieved by scanning the new subtree with the original subtree in parallel, and assigning new processes to the new subtree whenever a process is found in the original subtree.

During a backward move of a process, there might be occasions when, at the same time, a forward process executing the same part of the subtree is encountered. In those cases, care must be taken that nodes are not accidentally deleted so that the other process may not be able to proceed.

4.5. Backward Movement of a Process

When a subtree is reduced to true or false, the process traces the backward pointer to its parent node. In the case where the parent node is andpop or orpop where two processes were created, the transformation schemas call for a deletion of one of the branches. For example: the transformation schema $T! || \text{Prog}_1! \Rightarrow \text{Prog}_1!$ collapses the tree structure to the structure of $\text{Prog}_1$ only. Also, the process that was executing $\text{Prog}_1$ will be terminated and $\text{Prog}_1$ will be reassigned the process executing its parent node. Once $\text{Prog}_1$ is reduced to $\text{trueop}$ or $\text{falseop}$, the process will again move backward until its root has been reached.

A process may also move backwards when the predicate Return is invoked. Execution of the predicate Return may result in a deadlock if the variable to be
assigned also exists in the expression at the same time. For example: 

\[ x := [ [ 1, 2, 3 ], x ] \] results in a deadlock. Therefore, the expression to be assigned to the variable must be ensured that it does not contain the variable. This is achieved by a simple recursive procedure which traverses the structure of the expression to make sure that the variable is not contained in it. If the variable does not occur in the expression, the tree structure for the invocation of the predicate \textit{Return} is modified to the tree structure of \textit{wheress} as follows.

The process executing the \textit{wheress} node will automatically traverse backward. The only allowable nodes that the backward traversal may encounter are the sequential and parallel and's and or's, and the find nodes. No other nodes are allowed to be in the backward traversal path. This implies that the R-Maple program:

\[
\text{not } x := 4; \text{ Prog}
\]
is illegal since \textit{Return} cannot pass through a notop node. This restriction is valid because there is no good declarative reading for such a transformation. The transformation schemas described in Appendix C indicates that all resulting structures after the transformation results in having the bfind opcode.

4.6. \textbf{Free and Bound Variables}

Attention must be given to the free and bound variables in several instances. Most of them are related in the replacement of an expression to a variable. This may occur during the invocation of a user defined predicate or when the predicate \textit{Return} is being invoked and the process is about to discharge the find node. The other instances occur during the execution of the \texttt{where} node. Each will be discussed in detail.

When a user defines a predicate of the form

\begin{verbatim}
<def op ,predop ,Num_1, varop ,Num_2, Prog>,
\end{verbatim}

the predicate is assigned a unique numeric value for \texttt{Num} of 25 or over. All predicates defined by the system such as addition, subtraction, etc. are assigned numeric values of less than 25. This allows the interpreter to distinguish whether a call is made to a system defined predicate or user defined predicate. This distinguishing between system predicate and user defined predicates is necessary because when a user defined predicate is invoked, the R-Maple machine loads the body of the predicate from the system so that execution may proceed, whereas when a system predicate is invoked, no copying of any predicate body is made. Execution of a system predicate is immediate. That is, for example, invoking the
test for equality system predicate "Eq", the system checks for the operands supplied along with the invocation and the test is done immediately.

The tree structure for the body of the user defined predicate is stored in the system. The structure of \texttt{Prog} serves as a master copy of the body of the predicate. When the predicate is invoked, the body is copied to a working area so that the tree may be executed. The address for the root node of the body is stored in an array called \texttt{predicate_info} where the predicate number, \texttt{Num}_1, is used as an index to it. The variable number, \texttt{Num}_2, is stored in another array called \texttt{pred_output_var} with the predicate number, \texttt{Num}_1, also being used as an index. When the predicate is invoked by a call such as \texttt{<altrop,predop,Num}_1,Expr\texttt{>}, the location of the master copy of the body of the predicate is retrieved by searching the array \texttt{predicate_info} using the predicate number as index. The variable number for the argument is also retrieved from the array \texttt{pred_output_var}. The body of the predicate is then copied to a new location. The input argument, \texttt{Expr}, is assigned to the argument variable of the predicate. That is, for each occurrence of \texttt{<varop,Num}_2>, it will be replaced by a copy of the input expression \texttt{Expr}. This is in accordance with the transformation schema for the definition of predicates and the invocation of predicates in section 3.2. Hence, assigning an expression to a variable is equivalent to replacing the nodes for the variable by a copy of the expression. When assigning expressions to a variable in a tree structure, it is important that the variables that are free in an expression do not become bound after the expression is substituted in the tree structure. For example,
(find y in ...... x ......) \{x := .. y ..\}

should not result in the variable y being bound. In order to ensure this, the expression is first scanned to obtain a list of all the free variables in it. Then in the process of replacing the variables in the body of the predicate by the expression, whenever a find node is encountered and the variable that follows the find node is one of the free variables in the expression, all occurrences of that variable will be replaced by another variable in the subtree starting at the node where the find node occurs. Hence, for the above example, the tree structure is changed to:

(find yy in ...... x ......) \{x := .. y ..\}

Now the replacement of x may proceed to obtain:

(find yy in ...... yy .......).

Variables are not replaced in a tree structure if the parent node is a find node even if the variable that follows is the same as the variable to be replaced. That is no substitution is made for

(find x in ...... x ......) \{x := .. y ..\}

because the x which follows the find is a bound variable for its body. Hence no substitution is necessary for the entire subtree. Replacement of this kind also occurs during a backward movement of a process upon invocation of the predicate Return. Replacement of the variable takes place when the backward movement reaches the corresponding find node whereupon the find node is discharged. Refer to the transformation schema in Appendix C for more details.

Another instance when free and bound variables require special attention is during the backward move of \(bf\indss\), \(bf\indsp\), \(bf\indps\), and \(bf\indpp\) when
either an \textit{andsop} or an \textit{andpop} is encountered. In this case, the \textbf{and} structure is collapsed into a \textbf{where} structure. The transformation schema for \textbf{bfind} is shown below. Variables that are free in \texttt{Prog}_3 but are bound in \( \alpha \) must have their occurrences changed in \texttt{Expr} and \texttt{Prog}_1. Otherwise, the free variable in \texttt{Prog}_3 will be taken as bound variables after the transformation since the variables in \( \alpha \) are going to be discharged eventually in a \textbf{find} node. The transformation schema for a \textbf{bfind} construct with an \textbf{and} node as its parent node is shown below. The structure:

\[
\{\textbf{bfind} \alpha \text{ in } [ \ (m := \texttt{Expr} ) \text{ and } \texttt{Prog}_1 ] \text{ or } \texttt{Prog}_2 \} \text{ and } \texttt{Prog}_3
\]
is transformed to:

\[
\textbf{bfind} \alpha \text{ in } \{ [ \ (m := \texttt{Expr} ) \text{ and } ( \texttt{Prog}_1 \text{ and } \texttt{Prog}_3 ) ] \\
\text{ or } ( \texttt{Prog}_2 \text{ and } \texttt{Prog}_3 ) \}
\]

An example where care must be given to free and bound variables is when a structure like the following occurs.

\[
\{\textbf{bfind} z \text{ in } [ \ (m := \texttt{Expr} ) \text{ and } ( \texttt{trueop and } z := 1 ) ] \text{ or } \texttt{Prog}_1 \} \\
\text{ and } a := z + 1
\]

In the expression, \( a := z + 1 \), the \( z \) is bound to a \textbf{find} which is a parent of the \textbf{and}. If the structure is transformed according to the transformation schema without paying any attention to the binding of variables, the tree structure will become:

\[
\textbf{bfind} z \text{ in } \{ [ \ (m := \texttt{Expr} ) \text{ and } ( z := 1 \text{ and } a := z + 1 ) ] \} \\
\text{ or } \texttt{Prog}_1
\]

Note that the expression \( z := 1 \) has the variable \( z \) bound to a \textbf{find} that is already encountered during the backward move. This is indicated by the presence of \( z \) in the \( \alpha \) list of \textbf{where}. Now, when the \textbf{find} node for \( m \) is finally
reached, the find node will be discharged and the variables in α will be put into a new find node. The resulting structure is:

\[ \text{find } z \text{ in } z := 1 \text{ and } a := z + 1 \]

Notice that inconsistency arises for the binding of the variable \( z \) in \( a := z + 1 \). To ensure that the variables are correctly bound, variables that are free in the branch \( \text{Prog}_3 \) in Figure 1 but are bound in \( α \) must have their occurrences changed consistently to another variable in \( \text{Expr} \) and \( \text{Prog}_1 \). For the above example, the correct transformation should be:

\[
\{ \text{bfind } zz \text{ in } [ ( m := \text{Expr} ) \text{ and } ( \text{true and } zz := 1 ) ] \text{ or } \text{Prog}_1 \}
\text{and } a := z + 1
\]

which will eventually be transformed to as:

\[ \text{find } zz \text{ in } zz := 1 \text{ and } a := z + 1 \]

### 4.7. Hardware Consideration

The R-Maple interpreter was originally developed on a PDP 11/23 machine running Unix. Due to the slowness of the processor, the interpreter was ported to an Amdahl V8 running the MTS operating system. However, the compiler on the Amdahl machine could only handle data structure with a maximum size of 8K. The array which stores the tree structure of a R-Maple program exceeds this limitation. The development of the interpreter was finally ported to a VAX 11/780. Since the interpreter was written in C, there was relatively no problem in porting from one machine to another. All the above machines are uniprocessor systems. Therefore, the “processes” that are created during the execution of a

† Unix is a trademark of Bell Laboratories.
R-Maple program needs to share the same processor. This implementation of R-Maple was mainly to test out the feasibility of computation based on the rewriting rules as described earlier. No attempt is made in improving the efficiencies of the interpreter.

4.8. **Examples of R-Maple Abstract Program**

The following is a series of R-Maple programs written in concrete and abstract syntax, the latter of which can be executed by the interpreter. The first example is a demonstration of the use of parallel processes. The program basically tests whether the variable \(<\text{varop},1>\) is greater than 10 when at the same time, it tries to assign to it various values. The program backtracks when the value assigned is not greater than 10. The corresponding concrete syntax is as follows:

```
find x in (x > 10 || (x:=3 orp x:=8 orp x:=88));
print(x)
```

The abstract program that is accepted by the R-Maple interpreter is as follows:

```
findop varop 1 andsop andpop attrop predop 5 pairop varop 1 quoteop 10 orpop attrop predop 8 pairop varop 1 quoteop 3 orpop attrop predop 8 pairop varop 1 quoteop 88 attrop predop 7 varop 1.
```

The output of this program is:

```
88
```

```
```
The next example is a program to append two lists. This demonstrates the use of the *splitop* operation. The predicate is assigned a predicate number of 30. The concrete syntax is as follows (predicate name "Append" and variable names are used in the presentation instead of the use of predicate number and variable numbers.):

```plaintext
Append (argument) is
  find [lst,result ] = argument in
  find [lst 1,lst 2] = lst in
    if lst 1 = nil
      then result := lst 2
    else find [hd ,tl ] = lst 1 in
      find res 1 in
        Append ([[ tl ,lst 2 ]res 1 ]); result := [hd ,res 1 ]
```

The abstract syntax is:

```
defop predop 30 varop 1 splitop varop 2 varop 3 varop 1 splitop varop 4 varop 5 varop 2 ifop attrop predop 1 pairop varop 4 quoteop 0 attrop predop 8 pairop varop 3 varop 5 splitop varop 6 varop 7 varop 4 findop varop 8 andsop attrop predop 30 pairop pairop varop 7 varop 5 varop 8 attrop predop 8 pairop varop 3 pairop varop 6 varop 8.
```

The predicate can be invoked in a program of the form:

```plaintext
find x in Append ([[ [ 4,1,0 ], [ 2,0 ],x ] ]); print (x).
```

The corresponding abstract syntax is:

```
findop varop 10 andsop attrop predop 30 pairop pairop pairop quoteop 4 pairop quoteop 1 quoteop 0 pairop quoteop 2 quoteop 0 varop 10 attrop predop 7 varop 10.
```

The output is as follows:

```
[ 4 , [ 1 , [ 2 , 0 ] ] ]
```
The third example is a program that computes the Fibonacci Sequence. The predicate takes in an input argument specifying the nth Fibonacci number. The concrete syntax is shown below:

\[\text{Fib} \ (x) \ \text{is} \ \text{find} \ [m,n] = x \ \text{in} \]
\[\text{if} \ m = 0 \ \text{then} \ n := 2 \]
\[\text{else if} \ m = 1 \ \text{then} \ n := 3 \]
\[\text{else find} \ p \ \text{in} \ p := m - 1; \]
\[\text{find} \ q \ \text{in} \ q := m - 2; \]
\[\text{find} \ r \ \text{in} \ \text{Fib} \ ([p,r]); \]
\[\text{find} \ s \ \text{in} \ \text{Fib} \ ([q,s]); \]
\[\text{find} \ t \ \text{in} \ t := r + s; \]
\[n := t\]

That is, given a number \(n\), if \(n = 0\), its Fibonacci number is 2, if \(n = 1\), then its Fibonacci number is 3. Otherwise, the Fibonacci number of \(n\) is equal to the sum of the Fibonacci number of \(n - 1\) and \(n - 2\). The abstract program is as follows:

```
defop predop 30 varop 1 splitop varop 2 varop 3 varop 1 ifop attrop predop
1 pairop varop 2 quoteop 0 attrop predop 8 pairop varop 3 quoteop 2 ifop
attrop predop 1 pairop varop 2 quoteop 1 attrop predop 8 pairop varop 3
quoteop 3 findop varop 4 andsop attrop predop 10 pairop pairop varop 2
quoteop 1 varop 4 findop varop 5 andsop attrop predop 10 pairop pairop
varop 2 quoteop 2 varop 5 findop varop 6 andsop attrop predop 30 pairop
varop 4 varop 6 findop varop 7 andsop attrop predop 30 pairop varop 5
varop 7 findop varop 8 andsop attrop predop 9 pairop pairop varop 6 varop
7 varop 8 attrop predop 8 pairop varop 3 varop 8.
```

The predicate can be invoked as in the following program:

```
find x in Fib ([5,x]); print (x).
```

The abstract program is translated as:

```
findop varop 9 andsop attrop predop 30 pairop quoteop 5 varop 9 attrop
predop 7 varop 9.
```

The output is:
Although the above example does the job of computing the Fibonacci number, it requires double exponential time. A more efficient implementation can be as follows:

\[
\text{Fib}(y) \text{ is find [input, output ] } = y \\
\text{find } [x, \text{rest 1}] = \text{input in} \\
\text{find } [\text{max,rest 2}] = \text{rest 1 in} \\
\text{find } [\text{slast, last}] = \text{rest 2 in} \\
\text{if } x = 0 \text{ then output:=slast} \\
\text{else if } x = \text{max} \text{ then output:=last} \\
\text{else find } n \text{ in } n := \text{slast} + \text{last}; \\
\text{find } m \text{ in } m := \text{max} + 1; \\
\text{find } p \text{ in } \text{Fib } ([[x, m, last, n], p]); \\
\text{output:=p}
\]

This program computes the Fibonacci number by computing the Fibonacci sequence and storing the last two Fibonacci numbers found. The last Fibonacci number found is checked to see if it is the Fibonacci number required. Basically, the predicate is called with 4 input arguments and 1 output argument. The argument \( x \) is the \( n \)th Fibonacci number to be computed. \( \text{slast} \) and \( \text{last} \) are the last two Fibonacci number found. \( \text{max} \) indicates that \( \text{last} \) is the \( m \)th Fibonacci number generated and if \( \text{max} \) is equal to \( x \) then the program returns the result in \( \text{last} \). The abstract syntax is as follows:

defop predop 30 varop 12 splitop varop 6 varop 5 varop 12 splitop varop 1 varop 8 varop 6 splitop varop 2 varop 9 varop 8 splitop varop 3 varop 4 varop 9 ifop attrop predop 1 pairop varop 1 quoteop 0 attrop predop 8 pairop varop 5 varop 3 ifop attrop predop 1 pairop varop 1 varop 2 attrop
predop 8 pairop varop 5 varop 4 findop varop 10 andsop attrop predop 9 pairop pairop varop 3 varop 4 varop 10 findop varop 11 andsop attrop predop 9 pairop pairop varop 2 quoteop 1 varop 11 findop varop 13 andsop attrop predop 30 pairop pairop varop 1 pairop varop 11 pairop varop 4 varop 10 varop 13 attrop predop 8 pairop varop 5 varop 13.

The program can be invoked by the following statement:

\[
\text{find } x \text{ in } \text{Fib } ([ [ 5,1,2,3 ], x ]); \text{ print } ( x )
\]

It will compute the fifth Fibonacci number where the first two Fibonacci numbers are 2 and 3. The 1 in the input argument specifies that the maximum Fibonacci number known so far is the first one, namely 3. The abstract syntax is as follows:

\[
\text{findop varop 1 andsop attrop predop 30 pairop pairop quoteop 5 pairop quoteop 1 pairop quoteop 2 quoteop 3 varop 1 attrop predop 7 varop 1.}
\]

The output is the same as that shown in the last implementation.

The next example is another example to show the backtracking feature of R-Maple. It takes in an initial value as input and then it keeps on generating integers starting at the initial value until the resulting value is equal to 11. The program in concrete syntax is:

\[
\text{Gen } ( x ) \text{ is } ( \text{print } ( x ); x = 11 ) \text{ or } \\
\text{find } y \text{ in } y := x + 1; \text{ Gen } ( y ).
\]

The abstract program is:

\[
\text{defop predop 30 varop 1 orsop andsop attrop predop 7 varop 1 attrop predop 1 pairop varop 1 quoteop 11 findop varop 2 andsop attrop predop 9 pairop pairop varop 1 quoteop 1 varop 2 attrop predop 30 varop 2.}
\]

The predicate can be simply invoked by the abstract program:

\[
\text{attrop predop 30 quoteop 1.}
\]

The result is:
The final example is a program that performs the quicksort algorithm. Two predicates are defined. One is used to partition a list into two lists such that all the elements of the first list is less than the elements of the second list. The second predicate is used to append an unsorted list to a sorted list to produce the final sorted list. Therefore to sort an unsorted list, the call is to append the unsorted list to a nil sorted list to produce the sorted list. The concrete syntax of the predicate to partition two lists is presented first.

\textit{partition} (x) \textit{is}
find \[ input, res \] = x in
find \[ lst, num \] = input in
if \( lst = nil \) then \( res := [0,0] \)
else find \[ hd, tl \] = lst in
find r in partition (\[ [tl, num], r \]);
find \[ s, l \] = r in
if \( num < hd \) then \( res := [s, [hd, l]] \)
else \( res := [hd, s, l] \)

The abstract syntax for the predicate \textit{partition} is as follows:

\[
\text{defop predop 30 varop 1 splitop varop 2 varop 3 varop 1 splitop varop 4 varop 5 varop 2 ifop attrop predop 1 pairop varop 4 quotep 0 attrop predop 8 pairop varop 3 pairop quotep 0 splitop varop 6 varop 7 varop 4 findop varop 8 andsop attrop predop 30 pairop pairop varop 7 varop 5 varop 8 splitop varop 9 varop 10 varop 8 ifop attrop predop 3 pairop varop 5 varop 6 attrop predop 8 pairop varop 3 pairop varop 9 pairop varop 6 varop 10 attrop predop 8 pairop varop 3 pairop varop 6 varop 9 varop 10.}
\]

The concrete syntax for the \textit{append} is shown below:

\[
\text{append (x) is}
\text{find \[ input, output \] = x in}
\text{find \[ sorted, unsorted \] = input in}
\text{if \( unsorted = nil \) then \( output := sorted \)}
\text{else find \[ head, tail \] = unsorted in}
\text{find p in partition (\[ [tail, head],p \]);}
\text{find \[ s, l \] = p in}
\text{find m in}
\text{append (\[ [sorted, l], m]\]) ||}
\text{append (\[ [head, m], s, sorted\])}
\]

Two partitioned sublists \( s \) and \( l \) are sorted in parallel. This is the speeded version of quicksort where the concatenation of the two sorted sublists is done on the fly. The abstract program is:

\[
\text{defop predop 31 varop 1 splitop varop 2 varop 3 varop 1 splitop varop 4 varop 5 varop 2 ifop attrop predop 1 pairop varop 5 quotep 0 attrop predop 8 pairop varop 3 varop 4 splitop varop 6 varop 7 varop 5 findop varop 8 andsop attrop predop 30 pairop pairop varop 7 varop 6 varop 8 splitop varop 9 varop 10 varop 8 findop varop 11 andpop attrop predop 31 pairop pairop varop 4 varop 10 varop 10 varop 11 attrop predop 31 pairop pairop varop 6 varop 11 varop 9 varop 3.}
\]
The quicksort program can be invoked as:

```plaintext
find x in append ([ 0,[ 4,1,2,3,0 ] ],x ); print (x ).
```

The abstract program is:

```plaintext
findop varop 1 andsop attrop predop 31 pairop pairop quoteop 0 pairop quoteop 4 pairop quoteop 3 pairop quoteop 2 pairop quoteop 1 quoteop 0 varop 1 attrop predop 7 varop 1.
```

The output is:

```
[1, [2, [3, [4, 0]]]]
```

```
[1, [2, [3, [4, 0]]]]
```
Chapter 5
A Functional and Relational Programming Language

5. Motivation For A Functional and Relational Programming Language

R-Maple, as mentioned before, is a logic programming language with relation as its basic object. Hence, it lacks the expressive power of functions. For example, if a(x) is a function, it can be composed in the arguments of another function or relation such as b(1,a(x)). On the other hand, if “a” is a relation taking one input argument and generating an output value, then the predicate “a” needs to have two arguments such as a (x | y ) where x is the input argument and y is the output argument. Composition of relations is not so readily available. It can be achieved, however, by \( \exists y \{ a (x | y ) & b (1,y ) \} \). Some relations expressed in R-Maple can actually generate a unique result for each input argument. Therefore, in essence, they are functions. The disadvantage of expressing functions as relations is that extra information is kept for backtracking purposes which is never used in the case for functions. A purely functional programming language, on the other hand, does not have the non-deterministic characteristic of relations. Therefore, a combination of a functional programming language with a relational programming language can result with the best of the two worlds. R-Maple can be combined with a functional programming language to allow functions as basic objects, so that the readability of programs and the speed of execution may be improved. The language RF-Maple is the result of extending the features of a functional programming language called F-Maple to include the
features of R-Maple. The next two sections will describe the language F-Maple followed by the description of RF-Maple. Both of these sections are described in full details in [Voda & Yu 84]. The following is only an abstract from the paper.

5.1. Description of F-Maple

F-Maple (F stands for Functional) is a typed language. It provides not only for semantic extensibility, but also for syntactic extensibility. By this, it means that a user is allowed not only to define new types and functions but also he/she is allowed to design the syntax of the data types and the syntax of a function invocation. Although syntactic extensibility for data types has been explored by various researchers such as [Kand 83] and [Malu 81], F-Maple allows syntactic extensibility for functions as well. Examples in the following will demonstrate the ease of use and the extensibility nature of the language.

The basic types of F-Maple are number and string. From these basic types, a user can define new data types by means of generating productions. Generating productions are similar to the productions used in a context free grammar. The left hand side is a non-terminal specifying the name of the new data type. Non-terminals are capitalized while terminals are not. Non-terminals are used to indicate types while terminals, which are used only on the right hand side, contribute to the syntax of the data type. The non-terminals on the right hand side specifies what other data types the new data type is composed of. Terminals can be added to express the data type in a relatively readable structure. For example, the data type of a list of numbers Numlist can be defined as follows:

\[ \text{Numlist} \rightarrow \text{nil} \]
Section 5.1 Description of F-Maple

Numlist $\rightarrow$ head Number and tail Numlist

Symbols may be used as terminals in the generating productions. The data type for a complex numbers Complex can be defined as follows:

Complex $\rightarrow$ Number + Number $i$

The non-terminals occurring on the right hand side of the generating production can be F-Maple basic types or user defined data types.

Enumerated types can be defined by productions which do not contain any non-terminals on the right hand side. The type Bool is one example and can be specified by:

Bool $\rightarrow$ true

Bool $\rightarrow$ false

Parameterized types can be added into F-Maple. This is achieved by adding a parameter to the non-terminal on the left hand side of the generating production. For example, to define a generic type constructor List $(T)$ which is the type for a list whose elements are of type $T$, the following productions are needed:

List $(T) \rightarrow$ nil

List $(T) \rightarrow$ head $T$ and tail List $(T)$

Note that the definition for List $(\text{Number})$ is the same as for Numlist defined earlier. Another example is the generic type constructor for a binary tree whose nodes are of type $T$, Bintree $(T)$. It can be defined as follows:

Bintree $(T) \rightarrow$ empty

Bintree $(T) \rightarrow$

node $T$ left Bintree $(T)$ and right Bintree $(T)$

Values of a data type are generated from sentences produced from a non-terminal. For example, a data value of type Numlist is:
*head 1 and tail ( head 2 and tail ( head 6 and nil ) *)

This denotes a list of numbers containing elements 1, 2, and 6. Parenthesis play
a special role in data values in that they improve readability. For this reason,
they are not supposed to be used as terminals in the generating productions.
Another example of a data value of Complex type is:

\[ 42 + 35 \, i \]

and finally, an example of a data value of a generic data type Bintree (Number)
is:

\[
\text{node 5}
\text{left (node 2 left empty and right empty )}
\text{and right empty}
\]

It can be seen that the use of grammars in defining data types allows the
user to have complete control over the syntax.

### 5.2. Terms over F-Maple Types

Terms over F-Maple types are used to specify functions in F-Maple. Three
new kinds of term construct will be introduced. These are called function, case,
and variable productions. Each term has a type.

Function productions specifies the syntax of the function invocation and the
type of the arguments expected. They are distinguished from the generating pro­
ductions in that they will be written with \( \Rightarrow \) as the produces symbol. Hence,
example of function productions may be as follows:

\[
\text{Bool} \Rightarrow \text{search Bintree (Number ) for Number}
\text{Numlist} \Rightarrow \text{append Numlist after Numlist}
\]

Non-terminals used on the right hand side specify the types of formal arguments
while non-terminals on the left hand side specify the type of the function result. Thus, the first function takes two arguments, one of which is a binary tree whose nodes are numbers and the other is a number. The function returns a boolean value. The second function takes two formal arguments both of type Numlist, and the result of the function is also a Numlist. Function productions specify the data types for the function result and its arguments, and also the syntax of the function. Terms which are produced by the function productions in conjunction with the generating productions are transformed in such a way that the final term can be produced only by the generating productions. This implies the final term cannot be further reduced. For example, the term:

```
append nil after (head 5 + 7 and tail nil)
```

is produced from the second function production above and the generating production for Numlist. Computation of F-Maple reduces this term to the basic term:

```
head 12 and tail nil
```

Note that this term cannot be further reduced.

The case production is the main term construct used in the definition of the computation rule for functions in F-Maple. Case productions are of the following form:

```
S => case T of \( \alpha_1 \mid S \quad \alpha_2 \mid S \quad \ldots \alpha_n \mid S \)
```

where each \( \alpha_i \) is called a case label. The case production is legal if and only if the case labels correspond exactly to all the generating productions for the type
The computation rule for `append` may be specified as follows.

\[
\text{append } Ls_1 \text{ after } Ls_2 = \\
\text{case } Ls_2 \text{ of} \\
\text{nil | } Ls_1 \\
\text{head } H \text{ and tail } T \\
\text{head } H \text{ and tail } (\text{append } Ls_1 \text{ after } T)
\]

The variables `Ls_1` and `Ls_2` used in the body of `append` denote the two arguments expected during the invocation of the function. The variables are automatically declared by the addition of two new variable productions which are examples of the third term construct:

\[
\text{Numlist} \Rightarrow Ls_1 \\
\text{Numlist} \Rightarrow Ls_2
\]

Note that variables are also capitalized for readability purposes.

When `append` is invoked, `Ls_2` will be bound to the actual argument of type `Numlist`. Since a `Numlist` can be nil or can be a structure with head and tail value, the structure of `Ls_2` is tested in the case statement. In the case that `Ls_2` is a structure with head and tail, the head is given the name `H`, while the tail is given the name `T`. These variables can be used in the body of the production of the second case. Two new variables are declared in the second clause of case:

\[
\text{Number} \Rightarrow H \\
\text{Number} \Rightarrow L
\]

### 5.3. Examples of F-Maple Programs

An example of F-Maple program is the searching function of an ordered binary tree:

\[
\text{Bool} \Rightarrow \text{search } \text{Bintree } (T) \text{ for } T
\]

Its body can be defined as:
search Tree for Value =
  case Tree of
    empty | false
    node V left Lt and right Rt |
    case V < Value of
      true | search Rt for Value
      false |
    case Value < V of
      true | search Lt for Value
      false | true

The above definition presupposes the comparison functions, <, for each concrete type T used. For instance, for Bintree (Number), the following production is needed:

$$\text{Bool} \Rightarrow \text{Number} < \text{Number}$$

Another example of a F-Maple program is a function to reverse a number list:

$$\text{Numlist} \Rightarrow \text{reverse Numlist after Numlist}$$

Its body is:

$$\text{reverse L after M} =$$
  case L of
    nil | M
    head H and tail T |
      reverse T after (head H and tail M)

The reversed list of L appended to M is returned when the function is invoked.

5.4. Computation of F-Maple Programs

No attempt is made to include any explicit control mechanism in F-Maple. The computation is by lazy evaluation.

One point that needs to be mentioned is that the freedom of extending the syntax of data types and terms may lead to ambiguous grammars being
introduced into F-Maple. Rather than attempting to parse the basic values or terms, an interactive structure editor is used to prompt the user for the value of the type needed at any moment. The advantage is that this eliminates the need for the user to type in the long descriptive names as terminals because he simply enters the needed value to the production.

5.5. **Description of RF-Maple**

In combining a functional and relational programming language together, there is a choice of introducing functions in a relational environment, or introducing relations in a functional environment. In the first case, the standard predicate logic with functions as terms is obtained. The second case leads to a logic without formulas but only with terms. This kind of logic is called term logic.

In the design of RF-Maple, term logic is opted for. The advantage is that no change needs to be made to cater to functions in the new language. Relations are treated as functions with Boolean values. Since functions in a functional programming language have all of their arguments as input arguments only, relations without output arguments (which is equivalent to tests in R-Maple) can be simply introduced into a functional programming language. An extension needs to be made to a functional programming language, however, to permit Boolean functions with output arguments for those relations which have output arguments. The term “generators” will be used for functions with output arguments in RF-Maple.

Computations of RF-Maple is similar to the computations of R-Maple and F-Maple. Functions are computed by lazy evaluation of F-Maple and generators
by the rewriting rules of R-Maple. It should be noted that generators are slower to compute because of backtracking whereas functions execute without this overhead.

RF-Maple has four new constructs in addition to those of F-Maple. These are the parallel and, parallel or, assignment, and find constructs.

Conjunction and disjunction can be sequential or parallel. Sequential conjunction and disjunction can be predefined using the `case` construct as follows:

\[
Bool \Rightarrow Bool \; ; \; Bool
\]

\[
A \; ; B = \text{case} \; A \; \text{of} \; true \; | \; B \; \text{false} \; | \; false
\]

and

\[
Bool \Rightarrow Bool \; \text{or} \; Bool
\]

\[
A \; \text{or} \; B = \text{case} \; A \; \text{of} \; true \; | \; true \; \text{false} \; | \; B
\]

Parallel conjunction and disjunction can be introduced by two new productions:

\[
Bool \Rightarrow Bool \; || \; Bool
\]

\[
Bool \Rightarrow Bool \; \text{orp} \; Bool
\]

Control will be passed on to the two bodies as is the case in R-Maple.

Assignments are of the form:

\[
Bool \Rightarrow \alpha := T
\]

where \( \alpha \) is an identifier declared as \( T \Rightarrow \alpha \) for any type \( T \).

Find productions are of the form:

\[
Bool \Rightarrow \text{find} \; \alpha: \; T \; \text{in} \; Bool
\]

where \( \alpha \) is an identifier and \( T \) is a type. For each `find` production, two more productions are automatically added. These are the variable production \( T \Rightarrow \alpha \)
and the assignment production $\text{Bool} \implies \alpha := \text{T}$. The productions may be used in the body of $\text{find}$.

An example of a generator in RF-Maple is:

$$\text{Bool} \implies \text{generate Number from Bintree (Number)}$$

$$\text{generate } X \text{ from } \text{Tree} =$$

$$\text{case } \text{Tree} \text{ of }$$

$$\text{empty} \mid \text{false}$$

$$\text{node } V \text{ left } Lt \text{ and right } Rt \mid$$

$$X := V \text{ or generate } X \text{ from } Lt$$

$$\text{orp generate } X \text{ from } Rt$$

All $\text{find}$ productions are of the form:

$$\text{Bool} \implies \text{find } \alpha: \text{T} \text{ in } \text{Bool}$$

where $\alpha$ is an identifier and $\text{T}$ is a type. For each $\text{find}$ production, two more productions are automatically added. These are the variable production $\text{T} \implies \alpha$ and the assignment production $\text{Bool} \implies \alpha := \text{T}$. The productions may be used in the body of $\text{find}$. For example:

$$\text{find } X : \text{Number in}$$

$$\text{generate } X \text{ from}$$

$$\text{node 2 left node 8 left empty and right empty}$$

$$\text{and right node 5 left empty and right empty};$$

$$7 < X$$

This term reduces to true after one backtrack to obtain the value 8, thus satisfying the test $7 < X$.

Through the use of all eight productions, one can create arbitrarily complicated terms over any user defined types.
5.6. **Examples of RF-Maple Programs**

An example will be given for the RF-Maple implementation of parallel Quicksort. It is a generator of type \texttt{Bool}.

\[
\text{\texttt{Bool}} \Rightarrow \text{\texttt{sort Numlist into Numlist}}
\]

\[
\text{\texttt{sort \textsc{Il} into \textsc{Ol}}} =
\]

\[
\texttt{append already sorted (nil) after \textsc{Il} giving sorted \textsc{Ol}}
\]

The body of \texttt{sort} calls another generator \texttt{append}.

The definition of the generator \texttt{append} is a recursive one.

\[
\text{\texttt{Bool}} \Rightarrow
\]

\[
\texttt{append already sorted \texttt{Numlist} after \texttt{Numlist} giving sorted \texttt{Numlist}}
\]

\[
\texttt{append already sorted \texttt{SI} after \texttt{Ul} giving sorted \texttt{Ol}} =
\]

\[
\text{\texttt{case Ul of}}
\]

\[
\texttt{nil | \textsc{Ol} := \texttt{Sl}}
\]

\[
\texttt{head N and tail T |}
\]

\[
\text{\texttt{case partition T by N of}}
\]

\[
\texttt{small \texttt{Sml} and large \texttt{Lrg} |}
\]

\[
\texttt{find X: Numlist in}
\]

\[
\texttt{append already sorted \texttt{SI}}
\]

\[
\texttt{after \texttt{Lrg} giving sorted \texttt{X} |}
\]

\[
\texttt{append already sorted (head N and tail X) after \texttt{Sml} giving sorted \texttt{Ol}}
\]

Two partitioned sublists \texttt{Sml} and \texttt{Lrg} are sorted in parallel as in the example for the R-Maple version of quicksort presented in section 4.8.

The predicate above is a generator. The program \texttt{partition}, however, can be written as a function yielding two lists. The relevant definitions are as follows.

\[
\texttt{Pair} \rightarrow \texttt{small Numlist and large Numlist}
\]

\[
\text{\texttt{Pair}} \Rightarrow \text{\texttt{partition Numlist by Number}}
\]

\[
\text{\texttt{partition \textsc{Nl} by \textsc{Num}} =}
\]

\[
\text{\texttt{case \textsc{Nl} of}}
\]
Examples of RF-Maple Programs

The next example is a program that counts the nodes of a binary tree. The program is written as a function that returns the count. The function production is as follows:

\[
\text{Number} \implies \text{count nodes of Bintree (Number)}
\]

The definition is then:

\[
\text{count nodes of Tree = case Tree of empty } | 0 \quad \text{node V left Lt and right Rt | } 1 + \text{count nodes of Lt + count nodes of Rt (0)}
\]

The program can be easily converted to add up the value of all the nodes of the binary tree. This can be achieved by replacing the number 1 by the variable V in (0).

The next example is a generator which generates the Fibonacci number. It is written as a generator to demonstrate the use of the \textit{find} and \textit{assignment} construct.

\[
\text{Bool} \implies \text{fib of Number after Number and Number with Number yielding Number}
\]

\[
\text{fib of X after M and N with K yielding Y = case X = 0 of true | Y := M false | case X = K of}
\]
true | Y := N
false | find T: Number in fib of X after N and
M + N with K + 1 yielding T;
Y := T

The program assumes that the subtraction and addition function have been defined. The program uses the same algorithm as the corresponding program written in R-Maple in section 4.8. It keeps the last two Fibonacci numbers found and keep on generating a new Fibonacci number until the new number is the nth number required. In the program above, M and N are the last two Fibonacci numbers found and X is the nth Fibonacci number required. K indicates that N is the mth Fibonacci number. Hence when K equals X the program returns N. The generator can be invoked with the following statement.

```
find X : Number in
fib of 5 after 2 and 3 with 1 yielding X
```

This will generate the fifth Fibonacci number with the first two Fibonacci numbers as 2 and 3.

The final example appears also in the paper [Voda & Yu 84]. It illustrates cooperation of concurrent processes using partially instantiated streams as described in [Shap2 83]. The RF-Maple version of the queue manager will be presented.

Streams are means by which processes communicate. They are represented using lists. Messages are sent and kept in queues.

The queue manager accepts two kinds of messages. They are:

- **Message (T)** → **enqueue T**
- **Message (T)** → **dequeue T**
for putting and retrieving elements of type $T$ from the queue. As in Shapiro's program, the queue manager is communicating with two users via a merge process. The relevant types are:

\[
\begin{align*}
\text{Bool} &\Rightarrow \text{user 1 with stream List (Message (T))} \\
\text{Bool} &\Rightarrow \text{user 2 with stream List (Message (T))} \\
\text{Bool} &\Rightarrow \\
\text{merge List (T) with} \\
\text{List (T) yielding List (T)} \\
\text{Bool} &\Rightarrow \\
\text{queue with front List (T) end List (T)} \\
&\text{and messages List (Message (T))}
\end{align*}
\]

These are invoked by:

\[
\begin{align*}
\text{find } S_1, S_2, S_3 : \text{List (Message (T)) in} \\
&\text{user 1 with stream } S_1 \| \\
&\text{user 2 with stream } S_2 \| \\
&\text{merge } S_1 \text{ with } S_2 \text{ yielding } S_3 \| \\
&\text{find } Q : \text{List (T) in} \\
&\text{queue with front } Q \text{ end } Q \text{ and messages } S_3
\end{align*}
\]

The predicate bodies for \text{user 1} and \text{user 2} will not be given. \text{Merge} must be a primitive in RF-Maple. The body for \text{queue} is as follows:

\[
\begin{align*}
\text{queue with front } H \text{ end } T \text{ and messages } M = \\
\text{case } M \text{ of} \\
\text{nil | true} \\
\text{head } H_m \text{ and tail } T_m | \\
\text{case } H_m \text{ of} \\
\text{enqueue } V | \\
\text{find } N_t \text{ in} \\
\text{T}:= \text{head } V \text{ and tail } N_t \| \\
\text{queue with front } H \text{ end } N_t \text{ and messages } T_m \\
\text{dequeue } V | \\
V:= \text{head of } H \| \\
\text{queue with front (tail of } H ) \text{ end } T \text{ and messages } T_m
\end{align*}
\]

The first case terminates the cooperation of processes when the list of messages is
exhausted. Otherwise, the head of the list of messages contains the message \textit{enqueue} or \textit{dequeue}. In the first case, the end of the queue is partially instantiated with the pair composed of the value to be enqueued and the rest as yet not instantiated. The queue manager is then recursively invoked with the rest of the messages. In the latter case, \( V \) should be instantiated with the value at the head of the front. The queue manager then proceeds recursively with the rest of the messages. \textit{Head} and \textit{tail} are projection functions. \textit{Head} is defined as follows:

\[
\begin{align*}
T &\rightarrow \text{head of List (} T \text{)} \\
\text{head of } L &= \\
&\text{case } L \text{ of} \\
&\text{nil} \mid 0 \\
&\text{head } A \text{ and tail } B \mid A
\end{align*}
\]

Similarly for \textit{tail}. It is assumed that for each \( T \) used there is a production

\[
T \rightarrow 0
\]

giving a distinguished constant \( 0 \) of type \( T \). Note also that if the queue is empty when a dequeue message comes, \( H \), and thus \text{head of } H, \text{ will not be instantiated. Therefore, the user process trying to use V will be delayed when trying to determine the structure of V in a case construct. It will be allowed to proceed when an enqueue message from the second user arrives. Hence, it is necessary to restart the queue manager in parallel.}
6. **Background Of Semantics Theory**

Various methods have been proposed for describing the formal semantics of programming languages. The motives behind the need for formalizing the semantics of programming languages is that first, the notation used in a programming language may have a different meaning in another. For example, the meaning of an expression like $f(x)$ depends on the different rules about parameter passing in different languages [Stoy 77]. Second, computers require that programs presented to them follow a precise set of rules. The rules are often difficult to describe precisely and this is another reason for requiring great care and formality [Stoy 77]. Initially, the objective was to give a sufficiently precise description for implementers of the languages to construct a correct compiler. Nowadays, the emphasis is more on a description sufficiently precise for programmers to make rigorous statements about the behaviour of the programs they write. In the future, the most important benefit will probably be to language designers, who will be guided towards the design of better (cleaner) programming languages, with simpler formal descriptions [Stoy 77].

Three main methods for giving formal semantics description of programming languages will be described briefly in the following section. The semantics of RF-Maple will be described in yet another formal semantics in Chapter 7.
6.1. The Operational Approach

In this approach, the semantics of the programming language is defined in terms of the change of state of an "abstract" machine by each of its instructions. The idea is that although the abstract machine is probably completely unrealistic from a practical point of view, it is so simple that no misunderstanding can possibly arise with respect to its order code [Stoy 77]. The SECD machine described in [Hend 80] is an example of the definition of the LISP programming language by operational semantics. This approach does not fully define the semantics of a programming language because there is still the problem of rigorously defining the abstract machine. That is, giving a formal semantics for the machine code still needs to be done. In this sense, the operational approach seems to be pushing the problem of defining a programming language further into defining the abstract machine. Another problem with the operational approach as suggested by [Dijk 72] is that the method tends to be regarded as one which gives the result only of specific computations. By feeding the abstract machine with a set of input, it will generate the appropriate output. However, it is much more useful to be able to consider the class of all the computations that can possibly be evoked by the program, or even by a class of functions [Stoy 77]. This is in essence, defining a function such that for any input, the function will give the output which the programming language is intended to give. This leads to the second approach which has this objective in mind.
6.2. The Denotational Approach

Semantic valuation functions are used to map syntactic constructs in the program to the abstract values (numbers, truth, values, functions, etc.) which they denote [Stoy 77]. The definitions of semantic functions will be denotational, that is to say, structured so that the meaning of any composition phrase is expressed in terms of the meanings of its immediate constituents [Tenn 81]. A legal program is given a precise meaning as function from its inputs to its outputs. For example, the semantics of binary numerals may be formalized in the following table [Tenn 81].

Abstract syntax

<table>
<thead>
<tr>
<th>N ∈ Nml</th>
<th>binary numerals</th>
</tr>
</thead>
<tbody>
<tr>
<td>N ::= 0</td>
<td>1</td>
</tr>
</tbody>
</table>

Semantic domain

| N = {zero} + N | natural numbers |
| = {0,1,2,3 .... } |

Semantic function

| Nf : Nml → N |
| Nf [ [ 0 ] ] = 0 |
| Nf [ [ 1 ] ] = 1 |

The three sections of the table define (a) Nml, the syntactic domain of binary numerals whose syntax can be derived from the context free rule, (b) N, the semantic domain of natural numbers, and (c) the semantic function, Nf, using one "equation" for each of the possible forms of binary numeral [Tenn 81]. The problem of using models of denotational semantics with practical programs is that it is quite difficult to prove even simple properties of programs. The reason is that the denotational semantics stresses the model-theoretical part and underplays the importance of derivations within a formal theory [Voda 84].
6.3. **The Axiomatic Approach**

An axiom is associated with each kind of statement in the programming language, which states what one may assert after execution of that statement in terms of what was true beforehand [Stoy 77]. This approach grew out of Floyd's work [Floy 67] and the work of Hoare [Hoar 69]. An example of its use is as follows: consider a simple program that exchanges the values of two variables. Assume that before the program is executed, \( X \) and \( Y \) have some initial values \( X_0 \) and \( Y_0 \). It is desirable to prove that after the program is executed, \( X = Y_0 \) and \( Y = X_0 \). To achieve this, input and output assertions are included as comments in the program. The assertion \( X = X_0 \) & \( Y = Y_0 \) is included at the beginning of the program and the assertion \( X = Y_0 \) & \( Y = X_0 \) is included at the end of the program. These assertions are not to be executed by the program, but they serve as the way the programmer expects his program to behave. The essence of the approach is to generate from a commented program a set of statements called the verification conditions. If these statements and the input assertions are true, then the other assertions the programmer has put in his program are correct. Whereas the programmer's assertions are intended to hold only when control passes through the appropriate point, the verification conditions should be true in general, and they can be proved by a deductive system that knows nothing about sequential processing, loops, recursion, or other concepts about the flow of control of the particular program [Mann & Wald 77]. This approach has made an important contribution to the art of proving the correctness of programs. There is a minimum of new notation. However, this approach does not
clearly define computation.
7. **Theory of Pairs**

The methods for formalizing the semantics of a programming language described above seems to be either too complex or insufficient for the task. The difficulty in defining the semantics of a programming language using any of the above methods increases when the language allows assignments to variables. R-Maple is a declarative programming language which does not have any side effects and therefore, a weaker formal theory is sufficient in formalizing its semantics. Paul Voda [Voda 84] proposes to explain the semantics of a declarative programming language by means of two formal theories. One formal theory gives denotations to programs while the other one, which is actually a sub-theory of the first theory, specifies the operations of the computing machine executing the programs in that the computations correspond exactly to proofs in the weaker theory [Voda 84]. This formal theory is called the theory of pairs, TP. It is a first order theory with the power equivalent to the Peano Arithmetic which is a formal way of axiomatization of the natural number theory. As will be seen in the description of the theory, it is more suitable for programming languages because it is based on an equivalent of S-expression in LISP. The advantage is that S-expressions allow the encoding of data structure and programs in similar fashions.

The domain of the intended interpretation of TP consists of the individual 0 and is closed under the operation of pairing. Hence, 0, [0,0], [[0,[0,0]],0],
\[[0, [0,0]], [[0,0],0]\] are examples of individuals in the domain of TP which are denoted by literals. The term 0 denotes the individual 0. If the terms \(n\) and \(m\) denote the individuals \(n\) and \(m\) respectively, then \([n,m]\) denotes the ordered pair of \(n\) and \(m\). The numerals will be used as an abbreviation for terms as follows:

\[
\begin{align*}
0 & \equiv 0 \\
1 & \equiv [0,0] \\
2 & \equiv [0,[0,0]] \\
& \ldots
\end{align*}
\]

For the following discussion, \([a,b,c]\) will be written in place of \([a,[b,c]]\). A Lisp-like list composed of the elements \(a_1,a_2,\ldots,a_n\) will be written as

\[
[a_1,a_2,\ldots,a_n,0]
\]

Two pairs are equal if the corresponding components are equal and no pair is equal to 0. The atomic formulae in TP are \(s=t\) and \(s \subseteq t\) for any terms \(s\) and \(t\).

In addition to the logical symbols such as connectives and quantifiers of the first order predicate calculus, the regular first order logical axioms, TP includes the following axioms.

**Axioms of Identity:**

\[
\begin{align*}
x=x \\
x=x' & \land y=y' & x=y \rightarrow x'=y' \\
x=x' & \land y=y' & x \in y \rightarrow x' \in y' \\
x=x' & \land y=y' \rightarrow [x,y]=[x',y']
\end{align*}
\]

Also, TP includes the non-logical axioms of uniqueness of pairing, two axioms of membership and the axiom schema of induction.

**Non-Logical Axioms:**

\[
\begin{align*}
[x,y]=[x',y'] \rightarrow x=x' & \land y=y' \\
x \notin 0
\end{align*}
\]
\[
\begin{align*}
  x \in [y, z] & \leftrightarrow x = y \lor x \in z \\
  P(0) \land \forall x \forall y \{P(x) \land P(y) \rightarrow P([x, y])\} & \rightarrow \forall x P(x)
\end{align*}
\]

TP is a formal theory. The intended model for TP is satisfied in the domain of pairs. The model gives precise meaning to all functions and predicates introduced into TP. To formalize a functional programming language in TP, the language is defined as subclasses of terms of TP. Each function can be introduced as a conservative extension of TP by a new symbol satisfying its (recursive) definition. Similarly, a predicate of a logic programming language can be formalized in TP, and each predicate can be introduced into TP by a predicate symbol. The interpretation of TP then gives the denotation to the function or predicate.

By using the power of the predicate logic, TP gives meaning to a programming language. The control of the language is computed by rewrite rules which will be invariant under the meaning of programs being computed. In summary, TP gives meaning to programs. To formalize a programming language in TP, a subtheory S needs to be constructed which gives meaning to functions or relations in the language in that the proofs in S correspond exactly to computation. RF-Maple will be formalized in TP by setting up a sub-theory of TP whose proofs correspond exactly to the computations.

7.1. **Semantics of RF-Maple**

RF-Maple is a typed programming language. Introduction of types to a logic system like that of TP is not straightforward. In defining the semantics of RF-Maple, no typing theory will be introduced into TP. The semantics of RF-Maple
will be defined, instead, as a typeless language. This does not mean that the language to be defined in TP does not compute what RF-Maple does. Operationally, they are equivalent. The difference in defining RF-Maple as a typed language requires the introduction of types into TP formally, whereas, defining RF-Maple as a typeless language requires only a modification in the data structure used in the computation. The next section is a description of typeless RF-Maple. The language will be referred to as TRF-Maple.

7.2. Typeless RF-Maple

TRF-Maple differs from typed RF-Maple in that no new types can be defined in the language. The only allowable types in TRF-Maple are numerals and pairs. However, by manipulating the structure of a data value, say prepend a data value with a tag, the same result may be obtained like that of typed RF-Maple. For example, if every data value used in typeless RF-Maple is a pair, and that the head of the pair is taken as a tag which indicates the intended form of the data value in the tail of the pair, then, by the manipulation of the tag, a program written in typeless RF-Maple can simulate the effect that can be obtained in typed RF-Maple. Moreover, the operations of the typeless RF-Maple can be correctly defined semantically in TP, as will be seen in section 7.3. The difference between TRF-Maple and RF-Maple is that in TRF-Maple, the user has to be aware of the need to manipulate the tag to achieve the same result as that of typed RF-Maple (where types can be defined by generating productions, function productions, etc.).
Basically typeless RF-Maple is equivalent to RF-Maple in all aspects except
the following points. 1) The only allowable data values in typeless RF-Maple, as
mentioned above, are numerals and pairs. Hence, values such as \([1, [2, 3]]\),
\([[1, 2], [3, 4]]\) are legal values in typeless RF-Maple. 2) There are no more gen­
erating productions since no new types may be defined. 3) There are no more
function productions since all functions will return the same type. 4) There are
no more variable productions since all variables are of the same type also. 5)
Since there are no types defined in the language, the \texttt{case} construct is modified
so that the comparison is between two data values rather than on the data forms.
More specifically, the labels for the \texttt{case \textit{x} \ldots} statement are numbers
representing the tags (which are the intended data forms for the variable \textit{x}). The
comparison in the \texttt{case} statement is based on the head of \textit{x} with the numbers in
the labels. Since the head of \textit{x} is the tag for its data value in the tail, the com­
parison in the \texttt{case} statement is actually between tags (or intended data forms).
Also, the labels must be numbered in sequence starting with the label 0. That is,
the possible data forms for a variable or data value must be assigned tags in
sequence with the first data form numbered 0. 6) The \texttt{find} construct need not
specify the type of the variable to be bound. That is, the \texttt{find} construct can be
of the form:

\begin{verbatim}
find \alpha in \ldots
\end{verbatim}

7) the function \texttt{hd} and \texttt{tl} are system functions in TRF-Maple which select the
head or tail of a data value. These are not needed in R-Maple because R-Maple
provides the "split" operation which essentially allows the selection of head and
tail. In F-Maple and typed RF-Maple, a value can be mapped onto its data type in the label of the case construct and the parts of a data value can be readily retrieved. In TRF-Maple, the hd and tl functions need to be introduced as system functions to allow accessing of data in a pair. Finally, because functions can have functions as its arguments, the binding operator let is used which allows the function argument to be evaluated once, and prevents the function argument to be evaluated several times if it is needed in several places.

An example of a program written in TRF-Maple is the generator which generates a number from a binary tree of numbers. The program written in typed RF-Maple has been presented already in section 5.5.

\[
\text{generate } X \text{ from } \text{Tree} = \\
\text{let } Y := \text{Tree} \text{ in case } Y \text{ of} \\
0 \mid \{1,0\} \\
1 \mid X := Y.\text{tl.hd or} \\
\quad \text{generate } X \text{ from } Y.\text{tl.tl.hd or p} \\
\quad \text{generate } X \text{ from } Y.\text{tl.tl.tl}
\]

An empty tree structure represented in TRF-Maple can be:

\[
[0,0]
\]

where the first 0 is the tag indicating that the tree structure is empty. The tree structure with a data node and left and right branch represented in TRF-Maple can be:

\[
[1,\{\text{node},[\text{left, right}]\}]
\]

where left and right have a data structure either of the first form or the second form. The 1 indicates that the tree structure is non-empty, and that the tail of the whole structure contains the data values for the node and left and right
branch. The `let` binding operator ensures that the argument `Tree` to be evaluated only once. In case a function is passed to the argument `Tree`, the function argument will be evaluated once, and the result bound to the variable `Y`. The `case` operator selects the sub-program to be computed based on the comparison between the head of variable `Y` and the labels. If the head of `Y` is 0, then the data value for falsehood is returned (ie. `[1,0]`), else, the program assigns to `X` the value of the node (which is `Y.hd.tl`) and prepares for backtrack using the `or` operator. Notice that programs written in typeless RF-Maple is not as readable as their equivalent in typed RF-Maple. It is the user's responsibility to assign tags to his/her intended data forms. The data forms for truth and falsehood are often used in RF-Maple, therefore, the data value `[0,0]` will be used consistently to denote a true value, and `[1,0]` will be used to denote a false value. The constant `True` and `False` will be used to abbreviate the terms `[0,0]` and `[1,0].

Note that the tag for a binary tree with left and right branch is the same as the tag for the data value for falsehood. Tags in different types can be the same, therefore, it is the user's responsibility to check that correct intended typing is enforced. For example, when the above program is invoked, the variable `Tree` needs to have the intended form.

In essence, typeless RF-Maple forces the user to be aware of the manipulation of the tag whereas, in typed RF-Maple, type checking is left to the executing machine.
7.3. **Formalization of the Semantics of TRF-Maple in TP**

In formalizing the semantics of TRF-Maple, TP is rephrased as a term logic. Term logic [Herm 70] does not make a distinction between a term and a formula. This implies that logical connectives and quantifiers are just functions. The functions for sequential and parallel connectives will be introduced into TP.

All data values in TRF-Maple except numbers are prepended with a tag as an indication of the intended form. For each possible form of a data type, the first possible form is assigned a tag of value 0, the second 1, .. and so on. Hence, if the tag for the form `nil` is 0, the tag for `head Number and tail Numlist` is 1, then the data value `head 1 and tail head 2 and tail nil` represented in RF-Maple is represented as `[1,[1,[1,[2,[0,0]]]]]` in TRF-Maple. This is relatively cryptic in its form. Therefore, the following functions can be defined:

\[
\text{NIL} = [0,0] \\
\{X,Y\} = [1,X,Y]
\]

where `X` is intended to be a number and `Y` as a Numlist, either `NIL` or `\{X,Y\}`. Hence, the list above can be abbreviated by `{1,\{2,NIL\}}`. The use of the tag in the computation rules will become apparent when they are presented in the following.

The programs of TRF-Maple are a subclass of terms called TRF-terms. TRF-terms are composed from 0 and variables by pairing. `True ≡ [0,0]` will be used to indicate a true value, `False ≡ [1,0]` is taken as a false value. A list is always terminated by a `NIL`. Atomic TRF-formulae are composed of atomic formulae by connectives, negation, and existential quantifier. They are just func-
tions satisfying the following identities:

\[
\begin{align*}
[0,0] \lor [0,0] &= [1,a] \lor [0,0] = [0,0] \lor [1,a] = [0,0] \\
[1,a] \lor [1,d] &= [1,0] \\
[0,0] \land [0,0] &= [0,0] \\
[1,a] \land [0,0] &= [0,0] \land [1,a] = [1,a] \land [1,c] = [1,0] \\
\neg [0,0] &= [1,0] \\
\neg [1,a] &= [0,0]
\end{align*}
\]

where \(a, c,\) and \(d\) can be any RF-terms.

The **find** construct of TRF-Maple is introduced as the \(\exists\) function in TP. For any term \(a\), the term \((\exists x. a)\) denotes a function of all free variables in \(a\) but \(x\). \((\exists x. a) = \text{True}\) if there is a literal \(\pi\) such that \(a\{x:=\pi\} = \text{True}\) and \((\exists x. a) = \text{False}\) otherwise. All functions \(F(x)\) must be introduced into the theory of TP as one place functions in such a way that \(\models_{TP} F(x) = a\). All functions are evaluated using lazy evaluation. Lazy evaluation allows a delay in evaluating the arguments passed on to a function. The only disadvantage of lazy evaluation is that it is shallow. By this, it means that when the control reaches a pair for evaluation, it does not evaluate the pair but it reverses. This is not desirable on the outermost level where the evaluation would stop after one shallow reduction. For this reason, markers are introduced to force the full evaluation on the top level. Five markers are introduced into TP by the definitions to satisfy:

\[
\text{do}(x) = \text{wait}(x) = \text{down}(x) = \text{up}(x) = x
\]

The markers "do", "wait", "down", and "up" are called processes. They mark the positions where the computation takes place. The marker "do" goes down in a RF-term while the marker "wait" delays the computation of a term until another term is reduced. The marker "down" is used in the lazy evaluation of
function arguments. They are used to force the evaluation of all the arguments of a function. The marker "up" forces the full evaluation to the top level.

Functions $hd$, and $tl$ are introduced into TP with the obvious properties:

\[ \vdash_{TP} hd ([a,b]) = a \]
\[ \vdash_{TP} tl ([a,b]) = b \]
\[ \vdash_{TP} hd (0) = tl (0) = 0 \]

$hd (a)$ will be abbreviated to $a.hd$. Similarly $tl (a)$ will be abbreviated to $a.tl$.

Other functions that need to be defined are:

\[ x \| y = x \& y \]
\[ x \or x = x \lor y \]
\[ x := y = x = y \]

The case construct is introduced into TP to satisfy the following properties.

\[ \text{case } (0) = \text{case } (0,y) = 0 \]
\[ \text{case } ([0,a],y) = y.hd \]
\[ \text{case } ([m,n],a],y) = \text{case } ([n,a],y.tl) \]

The case construct of TRF-Maple are rewritten as case $(V,P)$ where $V$ is the tagged value on which the case decision is to be made and $P$ is the list of all sub-programs for each case arranged according to the increasing order of its corresponding tag value. For example, the case construct:

\[ \text{case } V \text{ of } 0 | Prog_1 \left[ 0,0 \right] | Prog_2 \]

is equivalent to the function case of the form:

\[ \text{case } (V.hd, [Prog_1,[Prog_2,\text{NIL}]] \) \]

If the tagged value $V.hd$ has a tag of 0, then the result is the sub-program corresponding to the first generating production of the data form of $V$. This is essentially the head of the list of sub-programs. Otherwise, the function is recur-
sively invoked with the tail of the tag $V.hd$ and the tail of the list of sub-
programs. Notice also, that the list of sub-programs is terminated by a NIL at
the end.

The operator **let** is introduced into TP to serve as an environment where
variables are allowed to be shared.

\[
\text{(let } v := a \text{ in } b ) = b \{ v := a \}
\]

This allows the evaluation of function using lazy evaluation. This is because the
evaluation of an argument in a function invocation can be delayed by assigning it
a variable in the **let** operator.

\[
f (g (2)) \Rightarrow \text{let } x := g (2) \text{ in } f (x)
\]

In the example, by using the **let** operator, evaluation of $g (2)$ will not take place
until the argument is needed in the evaluation of the body of $f (x)$. Once $g (2)$
is evaluated, its value is stored in the **let** operator so that other reference to
$g (2)$ can be immediately recovered.

By using the **let** operator, function argument needs to be evaluated once
even though several references of that argument are made. Also, the result from
a function inside the program can be bound to a variable using the **let** operator
so that it may be referred to in different places.

The backward shipment of assignments through enclosing disjunctions and
conjunctions by employing the associativity and distributivity of the connectives
until the assignment reaches the existential quantifier. For example:

\[
\{ (up (x := a) \& A) \lor B \} \lor C \Rightarrow
\{ up (x := a) \& A \} \lor (B \lor C)
\]
\[
\{ (\text{up}(z:=a) \land A) \lor B \} \land C \Rightarrow \\
\{ \text{up}(z:=a) \land (A \land C) \} \lor (B \land C)
\]

When the assignment reaches its quantifier, the quantifier is split for a possible backtrack and discharged in one side of disjunction.

\[
\exists x \{ (\text{up}(x:=a) \land A) \lor B \}\Rightarrow \\
\exists x (\text{up}(x:=a) \land A) \lor \exists x B \Rightarrow \\
\{ \text{let } x:=a \text{ in do } (A) \} \lor \exists x B
\]

The function Partition will be introduced into TP which partitions a list into a pair of lists by a number and returns the partitioned list is given here. The expanded name of the function defined in TRF-Maple, say \text{Partition } X \text{ by } N, \text{ is represented in the usual functional notation such as } \text{Partition} (X,N). \text{ In addition to the functions defined earlier to improve readability for tagged data values, (ie. NIL, \{x,y\}, etc.), the function } <x,y> \text{ is used to define a pair of Numlist } x \text{ and } y.

\[
\text{TP} \text{ Partition } (Nl,Num) = \\
\text{let } Y:=Nl \text{ in case } (Y,\langle\text{NIL},\text{NIL}\rangle, \\
\text{let } X:=\text{Partition } (Y.tl.tl,Num) \text{ in case } (X, \\
[ \text{case } (Num < Y.tl.hd.tl, \\
\langle X.tl.hd,\langle Y.tl.hd,X.tl.tl\rangle \rangle \\
\langle\{Y.tl.hd,X.tl.hd\},X.tl.tl > ] ) ] )
\]

The use of projection functions \text{hd} and \text{tl} are necessary to bypass the tag of the data value to access the value itself.

Notice that in the definition of the function \text{Partition}, the variable \text{X} is introduced in the \text{let} operator to refer to the result of the function invocation \text{Partition} (Y.tl.tl,Num). If the result is not shared on the outermost level among the sub-programs of the case construct, then the function needs to be invoked each time the value is needed. Also, the variable \text{Y} is used to bind to
the argument $NI$. In case $NI$ is passed a function as the argument, the result of
the function argument is bound to the variable $Y$ so that future reference to the
argument is readily available, and only one evaluation of the function is needed.

A sequence of existential quantifiers is abbreviated into the form $\exists \alpha A$ where
$\alpha$ is a sequence of variables. The empty sequence can be included by setting $\exists A$
to stand for $A$. and will be used to stand for either sequential or parallel con­
junction, similarly or.

The axioms and inference rules of TRF-terms can be specified as follows. However, they do not represent the only set of axioms and inference rules applicable in the computation of TRF-Maple programs. An implementation of TRF-
Maple may follow a whole new set of rewriting rules. The only requirement is
that there can be no ambiguity in the choice of which axiom or inference rule to
use in any circumstance during the computation.

a) **Initial Axioms** are of the form $A = \text{down}(do(A))$ for every RF-term $A$
not containing markers.

b) **Function Axioms** are of the form $F(x) = A$ where $A$ is a RF-term not
containing markers and having at most the variable $x$ free. It is required that
$\vdash_{TP} F(x) = A$.

c) **Computation Axioms** are the following ones for each RF-terms $A, B, C,$
$a, b, c,$ and $d$. The variable $v$ does not occur in the term $a$. $t$ will be used
to indicate the tag for the constructed value which is the intended data form
of the value. It should be noted again that True \( \equiv [0,0] \) is used to denote the true value in TRF-Maple, and False \( \equiv [1,0] \) is used to denote the false value in TRF-Maple.

**Turn-around Group.**

\[
do (v:=a) = (up(v:=a) \& True) \lor False
\]

provided that \( v \) is not free in \( a \).

**Identity Group.**

\[
do (0=[a,b]) = up(False)
do ([a,b]=0) = up(False)
do (0=0) = up(True)
do ([a,b]=[c,d]) = do (a=c) \& b=d
\]

**Negation Group.**

\[
do (\neg A) = \neg do (A)
\neg up(True) = up(False)
\neg up(False) = up(True)
\]

**Disjunction Group.**

\[
do (A \lor B) = do (A) \lor B
do (A \or B) = do (A) \or do (B)
up(True) \lor B = up(True)
up(False) \lor B = do (B)
up(False) \or B = B
\{\exists x (up(v:=a) \& A) \or B\} \or_1 C = \exists x \{up(v:=a) \& A\} \or_1 (\not B \or_1 C)
\]

\[
A \or B = B \or A
\]

In (1) \( \or_1 \equiv \or \) and \( \not B \equiv B \) unless \( \or_1 \equiv \or_1 \). In the latter case \( \or_1 \equiv \or_1 \) and if \( \or \equiv \lor \) then \( \not B \equiv do (B) \).
In (2) $B$ has one of the forms $\up (\text{True})$, $\up (\text{False})$, or $\exists \alpha \{ \up (v:=a) \text{ and } C \} \text{ or } D$ and $A$ is not of any of these forms. This prevents the computation to use this rule over and over again.

**Conjunction Group.**

\[
do(A \& B) = \do(A) \& B \\
\do(A \| B) = \do(A) \& \do(B) \\
\up(\text{False}) \text{ and } B = \up(\text{False}) \\
\up(\text{True}) \& B = \do(B) \\
\up(\text{True}) \| B = B \\
\{\exists \alpha \{ \up(v:=a) \text{ and } A \} \text{ or } B \} \text{ and } C = \\
\exists \overline{\alpha} \{ \up(v:=\overline{a}) \text{ and } (\overline{A} \text{ and } C) \} \text{ or } (B \text{ and } \overline{C}) (3) \\
A \| B = B \| A (4)
\]

where in (3) some of the bound variables in $\alpha$ must be renamed yielding $\overline{\alpha}$, $\overline{v}$, $\overline{a}$, and $\overline{A}$ should a free variable of $C$ become bound inside $\exists \alpha$. Also

$\overline{\text{and}} \equiv \text{and}$, $\overline{\text{or}} \equiv \text{or}$, $\overline{A} \equiv A$ and $\overline{B} \equiv B$ unless $\overline{\text{and}} \equiv \|$. In the latter case $\overline{\text{and}} \equiv \|$ and $\overline{\text{or}} \equiv \text{orp}$. If also $\overline{\text{and}} \equiv \&$ then $\overline{A} \equiv \do(A)$. Similarly, if $\overline{\text{or}} \equiv \lor$ then $\overline{B} \equiv \do(B)$.

In (4) $B$ has one of the forms $\up(\text{True})$, $\up(\text{False})$, or $\exists \alpha \{ \up(v:=a) \text{ and } C \} \text{ or } D$ and $A$ is not of any of these forms. This again prevents the computation to use this rule over and over again.

**Quantifier Group.**

\[
do(\exists v A) = \exists v \do(A) \\
\exists v \up(\text{True}) = \up(\text{True}) \\
\exists v \up(\text{False}) = \up(\text{False}) \\
\exists w \{ \exists \alpha \{ \up(v:=a) \text{ and } A \} \text{ or } B \} = \\
\exists \alpha \{ \up(v:=a) \text{ and } \exists w A \} \text{ or } \exists w B (5)
\]
Formalization of the Semantics of TRF-Maple

\[ \exists w \{ \exists \alpha \{ \up (v:=a) \text{ and } A \} \text{ or } B \} = \exists w, \alpha \{ \up (v:=a) \text{ and } A \} \text{ or } \exists w B \]  

(6)

In (5) and (6) the variable \( w \) is different from \( v \). In (5) the variable \( w \) does not occur in \( a \), whereas in (6) it does. The case when the variable \( w \) is the same as \( v \) is presented in the \text{let} rule.

**Case Group**

\[
do (\text{case } (x,P)) = \text{case } (\do (x),P) \\
\text{case } (\do ([a,b]),P) = \text{case } ([\do (a),b],P) \\
\text{case } ([\do (0),b],[P_1,P_2]) = \do (P_1) \\
\text{case } ([\do ([a,d]),b],[P_1,P_2]) = \text{case } ([\do (d),b],P_2)
\]

d) Replacement Rule.

\[
\begin{align*}
A = \cdots B \cdots, & \quad B = C \\
\hline
A = \cdots C \cdots
\end{align*}
\]

where \( B = C \) is a Computation axiom.

e) Forcing Group.

\[
\begin{align*}
down (\do (0)) & = \up (0) \\
down (\do ([a,b])) & = [\down (\do (a)),b] \\
[\up (a),b] & = [a,\down (\do (b))] \\
[a,\up (b)] & = \up ([a,b])
\end{align*}
\]

f) Projection Group.

\[
\begin{align*}
do (a.hd) & = \do (a).hd \\
do (a.tl) & = \do (a).tl \\
do ([a,b]).hd & = \do (a) \\
do ([a,b]).tl & = \do (b)
\end{align*}
\]

g) Termination Rules.

\[
\begin{align*}
A & = \up (B) \\
A & = B
\end{align*}
\]
Section 7.3  Formalization of the Semantics of TRF-Maple

h) **Function Call Rule.**

\[
\begin{align*}
\text{a} = \text{let } \alpha, \cdots \text{do } (f (b)) \cdots, \ f (x) &= c \\
\text{a} &= \text{let } \alpha, \cdots \ x:=b, \ \text{in } \cdots \text{do } (c) \cdots
\end{align*}
\]

where \( f (x) = c \) is a Function Axiom and all free variables of \( b \) have binding occurrences in \( \alpha \). In order to maintain the uniqueness of bindings the variable \( x \) has to be renamed to a new variable \( \overline{x} \) in the binding \( \overline{x}:=b \) and in the term \( c\{x:=\overline{x}\} \). In order to assure the uniqueness of derivations it is stipulated that the sequence of bindings \( \alpha \) is the shortest sequence binding all free variables of the term \( b \).

i) **Wait Introduction Rule.**

\[
\begin{align*}
\text{a} = \text{let } \alpha, u:=b \ \text{in } \cdots \text{do } (u) \cdots \\
\text{a} &= \text{let } \alpha, u:=\text{do } (b) \ \text{in } \cdots \text{wait } (u) \cdots
\end{align*}
\]

j) **Wait Elimination Rules.**

\[
\begin{align*}
\text{a} &= \text{let } \alpha, u:=\text{do } (0) \ \text{in } \cdots \text{wait } (u) \cdots \\
\text{a} &= \text{let } \alpha, u:=0 \ \text{in } \cdots \text{do } (0) \cdots
\end{align*}
\]

\[
\begin{align*}
\text{a} &= \text{let } \alpha, u:=\text{do } ([\alpha, \beta]) \ \text{in } \cdots \text{wait } (u) \cdots \\
\text{a} &= \text{let } \alpha, v:=a, w:=b, u:=[v,\overline{w}] \ \text{in } \cdots \text{do } ([v,\overline{w}]) \cdots
\end{align*}
\]

where in (7) the introduced variables \( v \) and \( w \) do not occur in \( \alpha \) and \( b \).

During the course of computation, several wait markers may be introduced. The sequence of wait elimination is last in last out. That is the latest wait gets eliminated first.

k) **Let Rules**

\[
\begin{align*}
\text{let } \alpha, x:=a \ \text{in } \cdots \text{do } (x:=a \ \text{in } b) \cdots \\
\text{let } \alpha, \overline{x}:=a \ \text{in } \cdots \text{do } (b) \cdots
\end{align*}
\]
where \( x \) is to be renamed to \( \overline{x} \) if \( x \) occurs free outside of \( b \).

\[
\begin{align*}
\text{let } \alpha \text{ in } & \ldots \exists v \{ \exists \beta (up(v := a) \text{ and } A) \text{ or } B \} \ldots \\
\text{let } \alpha, v := a \text{ in } & \ldots \exists \beta A \text{ or } \exists v \exists \beta B
\end{align*}
\]

\( \overline{A} \equiv A \) unless \( \text{ and } \equiv \& \). In the latter case \( \overline{A} \equiv do (A) \). Also, \( v \) is to be renamed to \( \overline{v} \) if \( v \) occurs free outside \( \overline{A} \) and \( \exists v \exists \beta B \).

\[
\begin{align*}
\text{let } \alpha \text{ in } & up(b) \\
up(b)
\end{align*}
\]

As a simple example, let us introduce the following functions by explicit definition into TP as:

\[
\begin{align*}
\vdash_{TP} f(y) = \text{case} (y, [\{0,0],y.tl], [0,y.tl.tl], \text{NIL}])
\end{align*}
\]

\[
\begin{align*}
\vdash_{TP} g(x) = \text{case} (x, [\{0,x.tl.tl], [\{0,0],x.tl.tl.hd], \text{NIL}])
\end{align*}
\]

This is a legal function because it takes in a value and simply returns another value. The steps of the computation of \( g (f ([\{0,0],\{0,\{0,0\}\}])) \) is partly given by the following proof. \( A \) will be used to abbreviate \( g (f ([\{0,0],\{0,\{0,0\}\}])) \).

\[
\begin{align*}
A = &\down (do (g (f ([\{0,0],\{0,\{0,0\}\}])))) \\
A = &\text{let } x := f ([\{0,0],\{0,\{0,0\}\}]) \text{ in } \\
&\down (do (\text{case} (x, [\{0,x.tl.tl], [\{0,0],x.tl.tl.hd], \text{NIL}])))
\end{align*}
\]

\[
\begin{align*}
A = &\text{let } x := f ([\{0,0],\{0,\{0,0\}\}]) \text{ in } \\
&\down (\text{case} (do (x), [\{0,y.tl.tl], [\{0,0],y.tl.tl.hd], \text{NIL}]))
\end{align*}
\]

\[
\begin{align*}
A = &\text{let } x := do (f ([\{0,0],\{0,\{0,0\}\}])) \text{ in } \\
&\down (\text{case} (\text{wait} (x), [\{0,y.tl.tl], [\{0,0],y.tl.tl.hd], \text{NIL}]))
\end{align*}
\]

\[
\begin{align*}
A = &\text{let } y := [\{0,0],\{0,\{0,0\}\}] \text{ in } \\
x := &\text{case} (do (y), [\{0,0],y.tl], [\{0,y.tl.tl], \text{NIL}])) \text{ in } \\
A = &\text{let } a := [0,0], b := [0,\{0,0\}], y := [a,b] \text{ in } x := case (do ([a,b])) \ldots \\
A = &\text{let } a := [0,0], b := [0,\{0,0\}], y := [a,b] \text{ in } x := case ([do ([a],b]) \ldots \\
A = &\text{let } a_1 := 0, a_2 := 0, a := [a_1,a_2], \ldots \\
\end{align*}
\]

\[
\begin{align*}
A = &\text{let } a_1 := 0, a_2 := 0, a := [a_1,a_2], \ldots \\
\end{align*}
\]
\begin{align*}
in \ x := & \text{case (}}\{(do (a_1, a_2)), \left[ \left[ 0,0 \right], y.tl \right], \left[ 0, y.tl.tl \right], \text{NIL} \}\}\cdots \\
A = & \text{let } a_1 := 0, a_2 := 0, a := [a_1, a_2], \cdots \\
& \text{in } x := \text{case (}}\{(do (a_2)), \left[ \left[ 0, y.tl.tl \right], \text{NIL} \}\}\cdots \\
& \text{A = let } a_1 := 0, a_2 := 0, a := [a_1, a_2], \cdots \\
& \text{in } x := \text{case (}}\{(do (0)), \left[ \left[ 0, y.tl.tl \right], \text{NIL} \}\}\cdots \\
A = & \text{let } a_1 := 0, \cdots x := do ([0, y.tl.tl]) \text{ in } down(\text{case (}\text{wait (}x\text{), } \cdots \\
A = & \text{let } \cdots x_1 := 0, y_1 := y.tl.tl, x := [x_1, y_1] \text{ in } \\
& down(\text{case (}do ([x_1, y_1]), \left[ \left[ 0, x.tl.tl \right], \left[ 0,0 \right], x.tl.tl.hd \right], \text{NIL}\}) \\
A = & \text{let } \cdots x_1 := 0, y_1 := y.tl.tl, x := [x_1, y_1] \text{ in } \\
& down(\text{case (}do ([x_1, y_1]), \left[ \left[ 0, x.tl.tl \right], \left[ 0,0 \right], x.tl.tl.hd \right], \text{NIL}\}) \\
A = & \text{let } \cdots x_1 := 0, \cdots \text{ in } down(\text{do } [0, x.tl.tl]) \\
A = & [0,0]
\end{align*}
8. **Future Work**

Considerable amount of improvement can be made to the implementation of R-Maple. For simplicity sake, the interpreter currently assigns expressions to variables using substitution of the expression. Invocation of predicates is by substitution also. This amounts to a lot of copying of tree structure during the execution especially when the expression to be assigned to a variable is large. In fact, during a trace of a program execution, most of the time is spent copying trees, scanning trees, etc. A more efficient improvement can be the use of environments. This allows the value assigned to variables to be retrieved and shared more efficiently. Also, some kind of administrative manager can be set up if subtrees are allowed to be shared. Currently, there is no sharing of subtrees allowed. If sharing is allowed, this will save considerably much time since fewer subtrees need to be copied.

In the current implementation, extra keywords are used for the abstract syntax which is not actually necessary. For example, the keyword *predop* in `<attrop,predop,...>`, is not necessary since the compiler that generates the code in abstract syntax is supposedly to be error free and the interpreter can assume that the number of the predicate to be invoked is supposed to follow the *attrop* opcode. The inclusion of these extra keywords for the current implementation is only for the sake of improved readability. Future implementation can improve in this regard. Also, the current implementation for the execution of the equality
test is not done in parallel. According to [Voda 83], during an equality test, the
tests for both branches of the two consed expression are to be done in parallel.
The current implementation checks each expression to see if there are any unas­
signed variables. If not, then the equality test for each node may proceed. Oth­
erwise, the process is blocked until the whole expression contains no variables.

So far, no compiler is available to translate a program written in concrete
syntax to abstract syntax. A front-end processor that will accomplish this task is
very desirable. Another point is that, although nondeterminism is available in
R-Maple, the actual implementation does not produce results that will reflect this
nature. If arguments are allowed to be entered interactively during execution,
some form of nondeterminism may result. R-Maple can also be made to allow
the use of any constants. In the current implementation, only numerals and pairs
are allowed. Also, a debugger will be useful during a trace of the execution of a
R-Maple program. Finally, if an algorithm is available to allow copying of tree
structure to be done more efficiently, this can significantly improve the perfor­
ance of the R-Maple machine.

RF-Maple is still under implementation. A working system will demonstrate
clearer the feasibility of the language and its ease of use. Again, the system can
bring out the nondeterministic nature of the language clearer if arguments are
allowed to be entered during execution.

Introduction of types into a formal logic system such as TP is still not clear
and straightforward. If types can be easily introduced into TP, then the seman­
tics of RF-Maple can be defined without having to convert the language into a
typeless one before the semantics can be defined.
9. Conclusion

It has been demonstrated that a logic programming language can be based entirely on the rewriting rules as described in section 3.4. Moreover, the rewriting rules correspond to a declarative reading in first order logic. The interpreter for R-Maple is considerably easy and straightforward to implement also because again, the operation of the interpreter is based on logic. R-Maple can be extended to include functions as can be seen from the description of RF-Maple. This allows some predicates which has the many-to-one correspondence between the input and the output to be encoded as a function. Functions can be executed more efficiently, since no backtracking is needed. Also, RF-Maple allows the user to define his or her own types according to his / her liking. Finally, it has been shown that TRF-Maple is semantically correct by its formalization using the Theory of Pairs. The semantics of R-Maple can also be said to be defined operationally.
References

[Abra 84]

[Berg & Kano 75]

[Colm 75]

[Cloc & Mell 81]

[Dahl & Samb 76]

[Dijk 72]
Dijkstra, E.W.: Talks on Programming Methodology; Advanced Course on Systems Architecture, Grenoble 1972 [memo EWD 356].

[Floy 67]

[Herm 70]

[Hend 80]

[Hewi 69]

[Hoar 69]

[Kand 83]
Kanda, A; Abramson, H. and Syrotiuk, V.: A Functional Programming Language Based on Data Types as Context Free Grammars; (submitted for publication), December 1983.
[Kowa]  

[Kowa 79]  

[Malu 81]  
Malusynski, J and Nilsson, J: A Notion of Grammatical Unification Applicable to Logic Programming Languages, Department of Computer Science, Technical University of Denmark, Doc, ID 967, August 1981.

[Mann & Wald 77]  

[Shap 83]  

[Shap 83]  

[Stoy 77]  

[Suss, Wino & Char]  

[Suss, McDe 72]  

[Tenn 81]  

[Voda 83]  

[Voda 84]  

[Voda & Yu 84]  
References

[Voda2 84]

[Warr 74]

[Warr 77]

[Warr1 77]
Appendix A

Concrete Syntax of R-Maple Numerals, Constants, Variables, Predicates and Expressions in BNF notation

The BNF rules for numerals are as follows.

\[
\text{Num} ::= \text{Digit} | \text{Num} \text{ Digit}
\]
\[
\text{Digit} ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

\[
\text{Ident} ::= \text{Letter} | \text{Ident} \text{ Letter} | \text{Ident} \text{ Digit}
\]
\[
\text{Letter} ::= a | b | c | d | e | f | g | h | i |
\]
\[
| j | k | l | m | n | o | p | q | r |
\]
\[
| s | t | u | v | w | x | y | z
\]

Constants are identifiers different from reserved words. Initially there is only one constant in R-Maple:

\[
\text{Const} ::= \text{nil}
\]

BNF productions for program predicates are as follows.

\[
\text{Pred} ::= \text{Cletter} | \text{Cletter} \text{ Digit} | \text{Cletter} \text{ Ident}
\]
\[
\text{Cletter} ::= A | B | C | D | E | F | G | I | J |
\]
\[
| K | L | M | N | O | P | Q | R | S |
\]
\[
| T | U | V | W | X | Y | Z
\]

BNF productions for expression are as follows.

\[
\text{Expr} ::= \text{Mexpr} | \text{Mexpr} + \text{Expr} | \text{Mexpr} - \text{Expr}
\]
\[
\text{Mexpr} ::= \text{Sexpr} | \text{Sexpr} \times \text{Mexpr} | \text{Sexpr} \div \text{Mexpr} | \text{Sexpr} \mod \text{Mexpr}
\]
\[
\text{Sexpr} ::= \text{Var} | \text{Num} | \text{Const} | [ \text{Expr} , \text{Exprs} ] |
\]
\[
\text{Const} ( \text{Exprs} ) | \text{Descr} | ( \text{Expr} )
\]
\[
\text{Descr} ::= . \text{Pred} | . \text{Pred} ( \text{Exprs} )
\]
\[
\text{Exprs} ::= \text{Expr} | \text{Expr} , \text{Exprs}
\]
Appendix B

Concrete Syntax of R-Maple Programs in BNF notation

Programs consist of decisions (if), searches (find), disjunctions (or, orp), conjunctions (";", "||"), negations (not), invocations, comparisons, assignments (:=), success (T), and failure (F).

\[
\begin{align*}
    \text{Prog} & ::= \text{Dprog} | \\
    & \quad \text{if Prog then Prog else Prog} | \\
    & \quad \text{find Decls in Prog} | \\
    \text{Dprog} & ::= \text{Cprog} | \text{Cprog or Dprog} | \text{Cprog orp Dprog} \\
    \text{CProg} & ::= \text{Sprog} | \text{Sprog;} \text{Cprog} | \text{Sprog} | \text{Cprog} | \\
    \text{Sprog} & ::= \text{T} | \text{F} | \text{not Sprog} | \text{Sprog} | \text{Atomp} | \text{Var:=Expr} | (\text{Prog}) \\
    \text{Atomp} & ::= \text{Invoc} | \text{ExprRelopExpr} \\
    \text{Relop} & ::= \text{eq} | \text{ne} | \text{lt} | \text{le} | \text{gt} | \text{ge} \\
    \text{Invoc} & ::= \text{Pred (Exprs)} | \text{Pred (Exprsopt|Var)} \\
    \text{Exprsopt} & ::= \text{Exprs} | \\
    \text{Decls} & ::= \text{Decl} | \text{Decl;} \text{Decls} \\
    \text{Decl} & ::= \text{Var} | \text{Svars:=Expr} | \text{Pred (Exprsopt|Svars)} \\
    \text{Svar} & ::= \text{Var} | [\text{Svar,Svars}] \\
    \text{Svars} & ::= \text{Svar} | \text{Svar, Svars} \\
\end{align*}
\]

The BNF production for concrete syntax of predicate definitions is as follows:

\[
\begin{align*}
    \text{Def} & ::= \text{Pred (Svars) is Prog} | \\
    & \quad \text{Pred (Svarsopt|Var) is Prog} | \\
    & \quad \text{Clauses} \\
    \text{Svarsopt} & ::= \text{Svars} | \\
\end{align*}
\]
Transformational Schemas for Computation of R-Maple Programs

Transformation for the invocation of predefined predicates Lt, Le, Gt, and Ge are as follows:

*Ordering Transformation* \((\text{Pred} \equiv \text{Lt}, \text{Le}, \text{Gt}, \text{Ge})\):

\[
\text{Pred}(\langle \text{pairop} , \langle \text{quoteop} , \text{Lit}_1 \rangle , \langle \text{quoteop} , \text{Lit}_2 \rangle \rangle)! \Rightarrow
\]

\[
\begin{cases}
\text{T} & \text{if } \text{Pred}^*(\text{Lit}_1,\text{Lit}_2) \\
\text{F} & \text{otherwise}
\end{cases}
\]

Transformations for test for equality are as follows:

*Identity Transformations*: \(\text{Expr}_1, \text{Expr}_2\) are not variables

\[
\text{Eq}(\text{Expr}_1,\text{Expr}_4)! \Rightarrow
\]

\[
\begin{cases}
\text{T} & \\
\text{F}
\end{cases}
\]

\[
\begin{cases}
\text{if } \text{Expr}_1^* = \langle \text{Expr}_2^*, \text{Expr}_5^* \rangle & \& \text{Expr}_4^* = \langle \text{Expr}_5^*, \text{Expr}_6^* \rangle \\
\text{Expr}_1^* = \text{nil} & \& \text{Expr}_4^* = \text{nil}
\end{cases}
\]

\[
\text{otherwise}
\]

\[
\text{Ne}(\text{Expr}_1,\text{Expr}_4)! \Rightarrow
\]

\[
\begin{cases}
\text{F} & \\
\text{T}
\end{cases}
\]

\[
\begin{cases}
\text{if } \text{Expr}_1^* = \langle \text{Expr}_2^*, \text{Expr}_5^* \rangle & \& \text{Expr}_4^* = \langle \text{Expr}_5^*, \text{Expr}_6^* \rangle \\
\text{Expr}_1^* = \text{nil} & \& \text{Expr}_4^* = \text{nil}
\end{cases}
\]

\[
\text{otherwise}
\]

Transformation for the invocation of the predicate Print as follows:

*Print Transformation*:

\[
\text{Print}(\langle \text{quoteop} , \text{Lit} \rangle)! \Rightarrow \text{T}!
\]

The transformation schema for arithmetic operations is as follows:

*Arithmetic Transformations* \((\text{Pred} \equiv \text{Add}, \text{Sub}, \text{Mul}, \text{Div}, \text{Rem})\):
Appendix C

\[
\text{Pred}(\langle \text{quoteop}, \text{Lit} \rangle | \text{Var})! \Rightarrow
\]
\[
\begin{cases}
\text{T}! & \text{where Var:=}\langle \text{quoteop}, \text{Lit}_1 \rangle \text{ if Pred}^*(\text{Lit}, \text{Lit}_1) \\
\text{F}! & \text{if } \exists a \text{ Pred}^*(\text{Lit}, a)
\end{cases}
\]

In the case of user defined predicates, invocation of predicates is transformed as follows:

\textit{Attribution Transformation} \quad \text{(Pred is defined in Defs)}:

\[
\text{Pred}(\text{Expr})! \Rightarrow \text{Prog}\{\text{Var:=Expr}\}!
\]

assuming the definition of Pred is

\textit{Pred (Var) is Prog}

Other unconditional schemas of transformation are:

\textit{Negation Scheduling}:

\[
\text{not }\text{Prog}! \Rightarrow \text{not } (\text{Prog}!)
\]

\textit{Conjunction Scheduling}:

\[
(\text{Prog}_1; \text{Prog}_2)! \Rightarrow \text{Prog}_1; \text{Prog}_2
\]

\[
(\text{Prog} \mid \mid \text{Prog}_2)! \Rightarrow \text{Prog}_1; \text{Prog}_2!
\]

\textit{Disjunction Scheduling}:

\[
(\text{Prog}_1 \text{ or } \text{Prog}_2)! \Rightarrow \text{Prog}_1 \text{ or } \text{Prog}_2
\]

\[
(\text{Prog}_1 \text{ orp } \text{Prog}_2)! \Rightarrow \text{Prog}_1 \text{ orp } \text{Prog}_2!
\]

\textit{Truth Table Transformations}:

\[
\text{not }\text{(T!)} \Rightarrow \text{F!}
\]

\[
\text{not }\text{(F!)} \Rightarrow \text{T!}
\]

\[
\text{T!; Prog} \Rightarrow \text{Prog!}
\]

\[
\text{T! \mid Prog} \Rightarrow \text{Prog}
\]

\[
\text{F!; Prog} \Rightarrow \text{F!}
\]

\[
\text{F! \mid Prog} \Rightarrow \text{F!}
\]

\[
\text{Prog \mid T!} \Rightarrow \text{Prog}
\]

\[
\text{Prog \mid F!} \Rightarrow \text{F!}
\]

\[
\text{F! or Prog} \Rightarrow \text{Prog!}
\]

\[
\text{F! orp Prog} \Rightarrow \text{Prog}
\]

\[
\text{T! or Prog} \Rightarrow \text{T!}
\]

\[
\text{T! orp Prog} \Rightarrow \text{T!}
\]

\[
\text{Prog orp F!} \Rightarrow \text{Prog}
\]

\[
\text{Prog orp T!} \Rightarrow \text{T!}
\]

\[
\text{find Var in T!} \Rightarrow \text{T!}
\]

\[
\text{find Var in F!} \Rightarrow \text{F!}
\]
Transformation for if-then-else is as follows:

\[ \text{Scheduling of Decisions:} \]
\[
(\text{if } \text{Prog}_1 \text{ then } \text{Prog}_2 \text{ else } \text{Prog}_3)! \Rightarrow \text{if } \text{Prog}_1! \text{ then } \text{Prog}_2 \text{ else } \text{Prog}_3
\]

\[ \text{Transformations of Decisions:} \]
\[\text{if } T! \text{ then } \text{Prog}_2 \text{ else } \text{Prog}_3 \Rightarrow \text{Prog}_2! \]
\[\text{if } F! \text{ then } \text{Prog}_2 \text{ else } \text{Prog}_3 \Rightarrow \text{Prog}_3! \]

Transformation for the split operation is:

\[ \text{Split Transformation:} \]
\[
\forall _1,\forall _2 \text{ not free in } \text{Expr}; \\
\text{Expr not variable} \\
(\text{find } [\forall _1,\forall _2]:=\text{Expr in } \text{Prog})! \Rightarrow \\
\begin{cases} 
\text{Prog}\{\forall _1:=\text{Expr}_1}\{\forall _2:=\text{Expr}_2\}! & \text{if } \text{Expr}^* = \langle \text{Expr}_1^*,\text{Expr}_2^* \rangle \\
F! & \text{otherwise}
\end{cases}
\]

Transformation for the find operation is as follows:

\[ \text{Search Scheduling:} \]
\[(\text{find } \forall \text{ in } \text{Prog})! \Rightarrow \text{find } \forall \text{ in } \text{Prog}! \]

Transformation for the predicate Return is as follows:

\[ \text{Return } (\text{Expr } | \forall ) \Rightarrow \text{bfindss } \alpha \text{ in } (\forall :=\text{Expr } ; T) \text{ or } F \]

Transformation for the \text{bfind} operation when a value needs to be moved back to its corresponding \text{find} is as follows:

\begin{align*}
\text{find } y \text{ in } (\text{bfind } \alpha \text{ in } (x:=e \text{ and Prog}_1) \text{ or } \text{Prog}_2) & \Rightarrow \\
(\text{bfind } \alpha, y \text{ in } (x:=e \text{ and Prog}_1) \text{ or } (\text{find } y \text{ in } \text{Prog}_2)) \\
\text{find } x \text{ in } (\text{bfind } \alpha \text{ in } (x:=e \text{ and Prog}_1) \text{ or } \text{Prog}_2) & \Rightarrow \\
(\text{find } \alpha \text{ in } \text{Prog}_1\{x:=e\} \text{ or } (\text{find } x \text{ in } \text{Prog}_2))
\end{align*}

where \text{Prog}_1 \equiv \text{Prog}_1 \text{ unless and } \equiv ; \ . \ In \ the \ latter \ case, \text{Prog}_1 \equiv \text{Prog}_1 !.

\begin{align*}
\{\text{bfind } \alpha \text{ in } (x:=e) \text{ and Prog}_1\text{ or Prog}_2\} \text{ or}_1 \text{Prog}_3 & \Rightarrow \\
\text{bfind } \alpha \text{ in } \{ (x:=e) \text{ and Prog}_1\} \text{ or } (\text{Prog}_2 \text{ or}_1 \text{Prog}_3)
\end{align*}

where \text{or}_1 \equiv \text{or } \text{and } \text{Prog}_2 \equiv \text{Prog}_2 \text{ unless } \text{or}_1 \equiv \text{orp} . \ In \ the \ latter \ case
or ≡ orp and if or ≡ or (sequential or), then \( \overline{Prog_2} ≡ Prog_2! \).

\[
\{ (bfind \alpha \in (x := e) \text{ and } \overline{Prog_1}) \text{ or } \overline{Prog_2} \} \text{ and } \overline{Prog_3} \Rightarrow \\
\overline{bfind \bar{\alpha} \in \{ \bar{x} := \bar{e} \} \text{ and } (\overline{Prog_1} \text{ and } \overline{Prog_3}) \} \overline{or_1} \\
(\overline{Prog_2} \text{ and } \overline{Prog_3})
\]

Some of the bound variables in \( \alpha \) must be renamed yielding \( \bar{\alpha}, \bar{x}, \bar{e}, \) and \( \overline{Prog_1} \) should a free variable of \( \overline{Prog_3} \) become bound inside \( \overline{bfind} \alpha \). \( \overline{and} \equiv \text{and}, \)

or ≡ or, \( \overline{Prog_1} ≡ Prog_1 \), and \( \overline{Prog_2} ≡ Prog_2 \) unless \( \overline{and_1} ≡ || \). In the latter case, \( \overline{and} ≡ || \) and \( or ≡ orp \). If also \( \overline{and} ≡ ; \), then \( \overline{Prog_1} ≡ Prog_1! \). Similarly, if \( or ≡ or \), then \( Prog_2 ≡ Prog_2! \).
Appendix D

Correspondence of Abstract Syntax with Logic

\[ \langle \text{trueop}, \text{nil} \rangle^* \equiv \forall x \; x = x \]
\[ \langle \text{falseop}, \text{nil} \rangle^* \equiv \exists x \; x \neq x \]
\[ \langle \text{attrop}, \text{predop}, \text{Pred}, \text{Expr} \rangle^* \equiv \text{Pred}^* (\text{Expr}^*) \]
\[ \langle \text{ifop}, \text{Prog}_1, \text{Prog}_2, \text{Prog}_3 \rangle^* \equiv (\text{Prog}_1^* \& \text{Prog}_2^*) \lor (\neg \text{Prog}_1^* \& \text{Prog}_3^*) \]
\[ \langle \text{findop}, \text{varop}, \text{Var}, \text{Prog} \rangle^* \equiv \exists \text{Var}^* \; \text{Prog}^* \]
\[ \langle \text{splitop}, \text{varop}, \text{Var}_1, \text{varop Var}_2, \text{Expr}, \text{Prog} \rangle^* \equiv \exists \text{Var}_1^* \exists \text{Var}_2^* (\langle \text{Var}_1^*, \text{Var}_2^* \rangle = \text{Expr}^* \& \text{Prog}^*) \]
\[ \langle \text{orsop}, \text{Prog}_1, \text{Prog}_2 \rangle^* \equiv \text{Prog}_1^* \lor \text{Prog}_2^* \]
\[ \langle \text{orpop}, \text{Prog}_1, \text{Prog}_2 \rangle^* \equiv \text{Prog}_1^* \lor \text{Prog}_2^* \]
\[ \langle \text{andsop}, \text{Prog}_1, \text{Prog}_2 \rangle^* \equiv \text{Prog}_1^* \& \text{Prog}_2^* \]
\[ \langle \text{andpop}, \text{Prog}_1, \text{Prog}_2 \rangle^* \equiv \text{Prog}_1^* \& \text{Prog}_2^* \]
\[ \langle \text{notop}, \text{Prog} \rangle^* \equiv \neg \text{Prog}^* \]
Appendix E

Abstract Syntax of R-Maple

Trueop and falseop have the following structure:

\[
\text{trueop} \quad \text{falseop}
\]

Quoteop and varop have the following structure:

\[
\text{quoteop} \quad \text{varop}
\]

\[
\text{Num} \quad \text{Num}
\]

Notop and pairop have the following structure:

\[
\text{notop} \quad \text{pairop}
\]

\[
\text{Prog} \quad \text{Expr}_1 \quad \text{Expr}_2
\]

Andsop and andpop have the following structure:
Orsop and orpop have the following structure:

Attrop and findop have the following structure:

Ifop and splitop have the following structure:
Whereop has the following structure: