R-Maple: A Concurrent Programming Language Based on Predicate Logic. Part I: Syntax and Computation.

Paul J. Voda

Department of Computer Science,
The University of British Columbia,
6356 Agricultural Road,
Vancouver, B.C. Canada V6T 1W5.

TR 83-9

1. Introduction.

Programs in modern programming languages consist of two components: *meaning* and *control*. Meaning of a program is given by a function or a predicate computed by the program. This function or predicate can be expressed as a term or formula of a formal logical theory. An interpretation of the theory assigns meaning to the terms and formulas. Thus the meaning of a program is the meaning of the corresponding term or predicate.

Control component is that part of programs which does not affect the meaning. Control provides instructions for the *machine* executing the program. Control directs the behaviour of the machine. Computation of programs can be viewed as an attempt by the executing machine to *prove* the program using certain axioms and rules of inference.

When a proof is found, i.e. the computation terminates, then the meaning component remains unaffected by the specific way (control) of the proof. On the other hand, when a proof is not found then the meaning component asserts only a *partial correctness*, of the program.

Some programming languages, especially the ones with explict parallelism, such as Algol-68 and Ada [18,10] contain a strong control component. The control in sequential programming languages as Algol-60 and Pascal [9,19] permits explicit sequencing of operations but no concurrent computation.

On the opposite end is Prolog [1,7] as a representative of *logic programming*. Conceptually at least, the control component is entirely absent in Prolog. The decision of how to sequence computations and what to do in parallel is left entirely to the Prolog interpreter acting as a virtual executing machine.

The meaning component of languages with strong control is quite weak. Complicated *meaning functions* of *denotational semantics* [15] are required to map the programs to formulas of a logical theory (lambda calculus). Although, it is easy to specify efficient computations in classical programming languages, it is quite cumbersome to prove properties of programs.

A very weak control in Prolog is outweighed by the direct connection of Prolog programs to formulas of predicate calculus. It requires a sophisticated interpreter of Prolog to assure a reasonably efficient execution of programs, but the proof of properties of programs is made easy. The only problem with the semantics of Prolog is that a programmer which writes programs by adding new axioms can easily render the whole system of axioms inconsistent. Now the close connection to logic is lost: one can prove *any* property of a program.

R-Maple (Relational Maple) is a programming language which tries to strike a balance between the meaning and control. Programs of R-Maple contain a control component and thus they cannot directly be formulas of predicate logic. However, the meaning function connecting programs in

This work was supported by the Natural Sciences and Engineering Research Council of Canada, grant A5098.

R-Maple to formulas of a logical theory is extremely simple and straight-forward.

The control component of R-Maple permits explicit specification of sequential and parallel computations. It allows the synchronization of parallel processes on the values of variables. Parallelism in functional programming languages is restricted to concurrent evaluation of function applications. R-Maple is based on relations and results may be non-deterministic in the sense that two parallel processes can come up with two different values satisfying a relation.

Programmers are not overly restricted in the way they write the programs in R-Maple. This includes programs which either do not terminate at all or terminate dead-locked. The naming function assigns a formula to all programs. Care is taken in the semantics to make sure that the intended meaning of nonsense programs cannot be *derived* in the formal theory of R-Maple. Since programs are not directly formulas, we can assure the consistency of the theory.

R-Maple is described in two parts. This part is concerned with syntax and computation rules. The naming function is also presented, but only Part II, devoted to semantics, investigates the conditions under which the intended meaning of R-Maple programs can be asserted. It also outlines a theory of *types* which simplifies the proofs that programs behave as they are intended to behave.

Although Part I does not deal with semantics, the reader does not need to be disappointed. The naming function gives him the intuition behind programs. He can prove partial correctness of programs, assuming them to terminate properly.

Nevertheless, Part I *entirely* includes, what is usually called, the *defining report* of a programming language. In addition to the defining report, we present a *rationale* behind the decisions to include single constructs of R-Maple. We also relate our constructs to similar constructs in another programming languages. We feel that by interleaving the formal and informal parts the readability of the report is increased.

How can we claim that we have a full defining report of a language without giving its semantics? The answer is perhaps surprising to everybody, but the hard core denotational semanticists. Defining reports traditionally define the semantics of programming languages by what is called the *operational* semantics. The operational semantics describes behavior of the machine which executes the programs. Part I fully defines the behaviour of R-Maple machine by presenting the complete set of its instructions in the form of *transformations*.

Even if we accept that a computation rule should not change the meaning of a program, the set of transformations operationally defining a programming language gives only a limited set of equivalences among programs. The transformations do no give any interpretation to programs. D. Scott [13] was the first one to point out that mere syntactic transformations are no basis for formal semantics. An interpretation of programs into a formal theory which posseses a model gives additional rules for proving properties of programs. Using only the identities given by transformations, we are not in position to prove even simple properties of programs.

For instance, given two functions:

$fac(n) =$ **if** $n = 0$ **then** 1 **else** $n \times fac(n-1)$
$facc(n,p) =$ **if** $n = 0$ **then** p **else** $facc(n-1, n \times p)$

the property $fac(n) = facc(n,1)$ can be proven only by induction. This property is a very important one because it permits a faster computation of factorials.

The denotational semantics uses Scott's models of lambda calculus as the basis for semantics. The above identity is proven by Scott's induction. The semantics of R-Maple uses the first order Peano arithmetic and subsequently the identity (or rather the one with predicates instead of functions) is proven with the help of proof by ordinary induction on n of

$$fac(n) \times m = facc(n,m)$$

Part I contains all of what a practical programmer expects from the definition of a programming language: The syntax is fully and formally specified. The execution of programs is precisely defined. As a bonus, we define the intended meanings of programs.

A description of a new programming language is usually welcomed with a slightly bored sigh: "Yet another one of those languages! It introduces a couple of new *features* and we are asked to wade through pages of boring syntax descriptions." To counter such objections we would like to sum up what we think is a novel approach in R-Maple:

1) R-Maple is based on relations rather than on functions. Thus it is not another classical programming language. It has explicit control over sequential and parallel processing. Relations permit additional non-determinism. R-Maple has a straight-forward and obvious meaning component. Actually, only such constructs have been included which map directly to logic.

2) Great care is exercised to show that the computation of R-Maple programs leaves the intended meaning intact. Whenever a relation permits more results, some of which are rejected by additional tests later in the program, *backtracking* must be invoked. This means that a different result is to be tried. Prolog has backtracking built into the interpreter. Consequently, one has only a dim perception of what is going on and one can only hope that no results are left untried. Computation rules of R-Maple make backtracking totally *visible* and one has confidence that all alternatives will be tried before failure is admitted.

3) The semantics of R-Maple is defined on *abstract programs* which have a simple structure but are not suitable for human readers. We also present a *concrete* syntax permitting a very readable form of programs. Our concrete syntax even covers the composition of relations. This escapes the constant need to invent new variable names for auxiliary results as in Prolog. Another very high level syntax sugaring of concrete syntax is that it permits both *procedural* and *clausal* form of definitions of predicates. Both the composition and definitions by clauses effectively combine the best features of functional and logic programs.

4) The relation between the abstract and concrete syntax is quite novel. Programs in concrete syntax are not translated into programs in abstract syntax. Concrete programs are mere *abbreviations* of abstract programs. As abbreviations, they do not have an independent existence and one does not need to show that the meaning is preserved by the transition. The abbreviations are specified in an exact way by *schemas of abbreviations*.

Let us outline the contents of Part I. The connection between computations and proofs is investigated in section (2). R-Maple is informally presented in section (3). Section (4) is concerned with formal logical principles underlying the design of R-Maple. The abstract syntax and the naming function is given in section (5). Some meta-theoretic properties of R-Maple programs are defined in section (6). Definitions of predicates are presented in section (7). Sections (8) through (11) introduce the concrete syntax of variables, predicates, expressions, programs, and definitions of predicates. Section (12) is concerned with the Prolog-like form of predicate definitions. Section (13) introduces the composition of relations. Section (14) presents the predefined predicates of R-Maple. The rules of computation are given in sections (15) through (17). Finally, section (18) gives various examples of computation and compares programs in R-Maple and Prolog.

The present author was greatly helped by long discussions he had with his colleagues Karl Abrahamson, Paul Gilmore, and Akira Kanda. Their contribution is gratefully acknowledged.

2. Computations as Proofs.

Let us have a closer look at the connection between computations and proofs. We have said that a computing machine uses certain axioms and rules of inference during the computation of a program. These do not necessarily include all axioms and rules of inference of a formal logical theory.

For instance, the computation of programs written in functional programming languages such as LISP or SASL [8,16] is based only on the β-rule of lambda calculus and on the Theorem of Equivalence.

The rule β is actually an axiom

$$(\lambda x.s)(a) = s[x:=a]$$

where $\mathbf{s}[x:=\mathbf{a}]$ stands for the term obtained from $\mathbf{s}$ by the substitution of $\mathbf{a}$ for all free occurrences of the variable x.

The Theorem of Equivalence permits us to replace identical terms or equivalent formulas in other terms or formulas without changing the meaning of the latter terms or formulas. The computation of a lambda-program replaces in a computed program an application $f(\mathbf{a})$, where $f = \lambda x.\mathbf{s}$, by the term $\mathbf{s}[x:=\mathbf{a}]$ to obtain a new program identical to the original one.

Control of functional programming languages does not give the executing machine great freedom in the way a proof is performed. Most programming languages insist on the *normal order* of computations where the leftmost and innermost function applications are removed first.

The computation in Prolog is based mostly on the transitivity of implication: When a program

$$\neg (A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, A_i \,\&\, \cdots A_n) \qquad (1)$$

is to be refuted, then in each computation step a properly instantiated axiom $B \rightarrow A_i$ is applied to obtain

$$\neg (A_1 \,\&\, A_2 \,\&\, \cdots \,\&\, B \,\&\, \cdots A_n) \qquad (2)$$

As a result we have (1) → (2). This process is repeated until a false formula **F** is reached. We have then (1) → **F**, i.e. ¬ (1). The special form of axioms in Prolog (Horn clauses) and the use of the transitivity of implications instead of the Equivalence Theorem contributes to the difficulties with negations in Prolog. If the sequence of implications reduces to **T** then nothing can be said about the original program (1).

An interpreter of Prolog has in theory, but not in practice, considerable freedom in choosing what part of (1) is to be replaced by what axiom. The interpreter could almost be a theorem prover. This freedom would necessarily slow down the execution of Prolog programs.

Finally, the "proofs" performed by machines executing programs in classical programming languages as Algol and Pascal can be viewed as based on a set of rewriting rules. Part of a program containing LHS of a rewriting rule is replaced by the RHS of the rule. The rewriting rules are usually quite arbitrary, similar to algorithms of Markov. This arbitrariness makes the connection to the meaning, i.e. to logic, difficult.

Computations in Prolog are the ones most closely connected to proofs in logic. But even so, the truly interesting proofs, the ones by induction, are not performed.

The computation of functional programming languages proves identity among terms. The computation in Prolog refutes formulas. Computation in R-Maple is performed on terms corresponding to certain formulas of predicate calculus. In this respect R-Maple is closer to Prolog. But unlike Prolog and like functional programming languages, the computation in R-Maple is based on the Equivalence Theorem. Part of a computed program is replaced by a program with the same meaning to yield a new program which has the meaning intact.

The Prolog machine has great freedom in applying the rules of computation. The R-Maple machine can apply a transformation only on certain positions within the program being computed. Whenever more transformations are applicable, the machine is free to choose any one. Unlike functional programming languages, R-Maple contains about one hundred different transformation rules. After all, R-Maple is a concurrent language with backtracking. Even so the rules are quite simple inasmuch as they are based on logical tautologies. The relatively large number of rules is necessary to assure the efficiency of computations.

3. Informal Introduction to R-Maple.

R-Maple is a relational programming language without side effects caused by state changes. Relational - as opposed to functional - means that the basic objects are relations. Functions have at most one result for each value of arguments. Relations can have more "result" values satisfying the same arguments. The possibility of multiple results brings into R-Maple *non-determinism* which is not present in functional programming languages.

Although we speak of variables and, in a figurative sense, of assignments to variables, R-Maple is an *applicative* language. Perhaps a better term is *attributive* because relations are attributed, rather than applied, to attributes. Once a variable obtains a value by a *substitution* the value is never changed again.

R-Maple is so closely connected to predicate logic that it can be called a *logic programming* language. Programs of R-Maple correspond to formulas of predicate calculus. Computation of a program corresponds to a proof of a formula.

Formulas of predicate logic are constructed from terms which can contain individual variables. Terms occur in atomic formulas. Atomic formulas are connected by logical connectives and quantifiers. Programs of R-Maple are constructed from *expressions* which can contain *program variables*. Expressions occur in *invocations*. Invocations are connected by *connectives* and *searches*. The following table summarizes the correspondence:

R-Maple:	Logic:	R-Maple:	Logic:
program	formula	expression	term
prog. variable	indiv. variable	invocation	atomic formula
search	$\exists$ quantifier	connective	connective

Every program of R-Maple names a formula of predicate logic. The meaning of the named formula gives the *intended meaning*, or partial correctness, of the program. Two or more programs with the same meaning can differ in the *control* component. The control component of a program directs the proof of the program when it is computed by R-Maple machine.

The invocation $P(x)$ invokes the *program predicate* P with the argument x. P may be a *predefined* or *defined* (program) predicate. The invocation $Eq(4,5)$ which can also be written as 4 **eq** 5 invokes the predefined predicate *Eq*. This invocation will be transformed by computation into the program **F** naming falsehood. The invocation [4,5] **eq** [4,5] transforms to truth: **T**. Braces "[ ]" enclose pairs. Predicates which reduce to **T** or **F** are called tests. The program 4 **eq** 5 names the the atomic formula $4 = 5$.

The invocation $Add(3,5 \mid x)$ contains two input arguments 3, 5 and the *output* variable x. *Add* is a predefined *generator* and when computed it transforms into an *assignment* to its output variable: x:=8. $Add(3,5 \mid x)$ names the formula $8+5 = x$; this is equivalent to $x = 8$ which is named by the assignment x:=8.

Tests and generators are connected by connectives. **A**; **B** is a *sequential and*. First the program **A** is computed. When it reduces to **T** then the program **B** is computed and its result is the result of the whole program. If **A** reduces to **F** then the whole program reduces to **F**. **A** || **B** is a *parallel and*. Computations of both operands are performed in parallel. This time **B** can reduce to **F** thus reducing the whole program to **F**. The other connectives are negation **not** , and sequential and parallel disjunction: **or** , **orp** . The computation of these connectives uses the corresponding truth tables of propositional logic. Conjunctions and disjunctions group to the right; conjunctions bind stronger than disjunctions.

The program

x eq 8 || 6 lt 9 or not 5 eq 6 (1)

names the formula $x = 8 \;\&\; 6<9 \vee \neg\, 5 = 6$. Both sequential and parallel disjunctions name the disjunction " $\vee$ ". Conjunctions name " & ". The program (1), when evaluated, reduces to **x eq 8 or not 5 eq 6**. The computation is then *delayed* on the identity test **x eq 8** until a parallel program *substitutes* a value for the variable x. If we replace the sequential disjunction in the program (1) by a parallel **orp** then the program (1) reduces to **T** even though the value of x is not yet known.

A frequent occuring combination **A; B or not A; C** can be shortened to the familiar form of decision

If A then B else C

This **If** statement, obviously, names the formula **A & B** $\vee \neg$ **A & B**. **If** statements are more

efficient to execute than the explicit forms since the test **A** is computed only once.

Generators and tests can be put into the scope of a search:

find x **in** $Add(6,7 \mid x)$; x **lt** 15 (2)

This search names the formula $\exists x(6+7 = x \ \& \ x<15)$. Since the computation in R-Maple always leaves the meaning *invariant*, we can expect the execution of (2) to yield **T**.

The execution of (2) will start by the execution of the predefined generator *Add* leading to

find x **in** x:=13; x **lt** 15

Now the transformation of successful search

find x **in** x:=**s**; $P(x) \Rightarrow P(\mathbf{s})$

is applied to obtain "13 **lt** 15" and from there **T**. The transformation of succesful search is based on the logical tautology

$\exists x(x = \mathbf{s} \ \& \ P(x)) \leftrightarrow P(\mathbf{s})$

provided that the variable x does not occur free in **s**.

Tests (*Even*) and generators (*Succ*) are defined by predicate definitions:

Even(x) **is find** r **in** $Rem(x,2 \mid r)$; r **eq** 0
Succ($x \mid next$) **is** $Add(x,1 \mid next)$

These definitions name the formulas

$\forall x(\ Even(x) \leftrightarrow \exists r(\ Rem(x,2,r) \ \& \ r = 0\ )\ )$ (3)
$\forall x(\ Succ(x,next) \leftrightarrow Add(x,1,next)\ )$

Note that the difference between tests and generators, so important on the level of control, completely disappears on the level of meaning. Incidentally, (3) is equivalent to

$\forall x(\ Even(x) \leftrightarrow \exists r Rem(x,2,0)\ )$

The invocation *Even*(8) is computed as

Even(8) => **find** r **in** $Rem(8,2 \mid r)$; r **eq** 0 => **find** r **in** r:=0; r **eq** 0 => 0 **eq** 0 => **T**

Generators can have multiple results.

$G(\mid x)$ **is** x:=3 **or** x:=5 **or** x:=8

Generators with multiple results cause *backtracking* when a generated value fails to satisfy a subsequent condition:

find a **in** $G(\mid a)$; a **gt** 6 (4)

First the invocation of G is computed:

find a **in** (a:=3 **or** a:=5 **or** a:=8); a **gt** 6

As a part of the execution of a:=3, and in the preparation for possible backtracking, the sequential *and* will be distributed over the sequential *or*. This transformation preserves the meaning since it relies on the distributivity of logical connectives.

find a **in** a:=3; a **gt** 6 **or** (a:=5 **or** a:=8); a **gt** 6 =>
(**find** a **in** a:=3; a **gt** 6) **or** (**find** a **in** (a:=5 **or** a:=8); a **gt** 6)

The last step relies on the logical tautology:

$\exists x(\ A(x) \vee B(x)\ ) \leftrightarrow \exists x A(x) \vee \exists x B(x)$

The computation continues in familiar way:

3 **gt** 6 **or** (**find** a **in** (a:=5 **or** a:=8); a **gt** 6) =>
F or (**find** a **in** (a:=5 **or** a:=8); a **gt** 6) =>
find a **in** (a:=5 **or** a:=8); a **gt** 6

The first result was unsucessfully tried; the computation is just about to distribute the

conjunction again in order to try out the second result $a:=5$. This fails and we shall have

find a **in** $a:=8$; a **gt** $6 \Rightarrow$ **T**

Had we replaced the test a **gt** 6 by the test a **eq** 6 in the program (4) then all three alternatives would fail thus failing the whole program.

Tests and generators can be recursive

Range(*low*,*high* | *x*) **is**
low **le** *high*; (*x*:=*low* **or** (**find** *ll* **in** *Succ*(*low* | *ll*); *Range*(*ll*,*high* | *x*)))

Range successively generates all values x in the interval $low \leq x \leq high$.

The intended meaning of the predicate *Range* can be proven by induction to be equivalent to:

$Range(low,high,x) \leftrightarrow low \leq x \;\&\; x \leq high$

The nuisance of invention of names for auxiliary variables, like *ll* in the above example, will be overcome by a neat syntax sugaring permitting the composition of relations. The generator *Range* can be written in a compact form:

Range(*low*,*high* | *x*) **is** *low* **le** *high*; (*x*:=*low* **or** *Range*(*low*+1,*high* | *x*))

Split search is the last primitive construct of R-Maple:

find [*hd*,*tl*]:=*list* **in** *P*(*hd*,*tl*) (5)

Split searches permit us to break apart pairs and find the corresponding values for pairs of variables. For instance, if the list *lst* obtained the value [6,*lst*2] then the execution of (5) would lead to the execution of *P*(6,*lst*2).

The split search (5) names the following formula.

$\exists hd \exists tl([hd,tl] = list \;\&\; P(hd,tl))$

4. Formal Principles of R-Maple.

R-Maple programs operate on natural numbers which can be also viewed as pairs composed of natural numbers or another pairs. An R-Maple program can be viewed from three aspects:

1) As a sequence of symbols with a fairly rich syntactic structure which makes the program readable and easy to understand.
2) As a sequence of symbols composed in a hierarchical way which is less readable by humans but may be easily composed and decomposed by other program. Such a program is encoded as data.
3) As an abstract object of certain properties. These properties can then be asserted in order to ascribe meaning to the program.

Programs in the first form are in the theory of programming languages said to be in *concrete syntax*. Programs of the second form are in *abstract syntax*. Programs in the third form are said to be *denoted* by programs in the first two forms.

We shall define programs of the first two forms to be terms in a formal theory. Programs of the third form will then be objects from the universe of an interpretation of the theory. Since programs are to operate on natural numbers and also be data for other programs, the abstract objects will have to include natural numbers. Although it may come as a surprise, we do not need more than natural numbers. We shall manage without additional data structures and even without functions of the lambda calculus variety. The formal theory in which the programs of the first two forms are terms will be the most common of all theories, first order Peano arithmetic.

The notation and terminology of formal arithmetic used in this paper is based on two basic logical texts: Kleene's Introduction to Meta-mathematics, and Schoenfield's Mathematical Logic [6,14]. We shall, however, slightly modify the symbols for individual variables and for predicates. Instead of traditional and rather dry symbols as $x,y,z,\ldots$ and $F,G,H,\ldots$ we shall use identifiers of

programming languages. Identifiers will start with capital or small letters and consist of digits or small letters. Capitalized identifiers will be used as symbols for predicates whereas small letter identifiers will be individual variables. For reasons which will become clear shortly, individual variables, constants, and predicate symbols of formal arithmetic will be underlined. Thus $\underline{x}$, $\underline{product}$, $\underline{result}$,... are individual variables ranging over natural numbers. We shall employ only one-place predicate symbols like $\underline{Append}$, $\underline{Reverse}$, $\underline{G}$,.... Numerals of arithmetic will be in italics: $1,2,3,\cdots$ Numerals $1,2,3,\cdots$ are introduced as abbreviations for $0',0'',0''',\cdots$ respectively.

We shall work in a recursive extension of arithmetic where, the operation $\binom{n}{2}$ satisfying

$$\binom{0}{2} = 0$$
$$\binom{n'}{2} = \binom{n}{2} + \underline{n}$$

has been introduced. We can now introduce the constant $\underline{nil}$ and the operation of pairing by explicit definitions:

$$\underline{nil} = 0$$
$$<\underline{a},\underline{b}> = \binom{\underline{a}+\underline{b}+1}{2} + \underline{a} + 1$$

Note that that the operation "$<,>$" satisfies the uniqueness property expected from a pairing operation. Also note that due to the factor $+1$ in the definition, no pair is equal to $\underline{nil}$. Both components of a pair are lesser numbers than the number encoding the pair. Every number either encodes a pair or is $\underline{nil}$. We thus have:

$$<\underline{a},\underline{b}> = <\underline{a}',\underline{b}'> \leftrightarrow \underline{a} = \underline{a}' \;\&\; \underline{b} = \underline{b}' \tag{1}$$
$$<\underline{a},\underline{b}> \neq \underline{nil} \tag{2}$$
$$\underline{a} < <\underline{a},\underline{b}> \tag{3}$$
$$\underline{b} < <\underline{a},\underline{b}> \tag{4}$$
$$\underline{a} = \underline{nil} \vee \exists \underline{b} \exists \underline{c}(\underline{a} = <\underline{b},\underline{c}>) \tag{5}$$

Programs in the abstract syntax are defined as n-tuples of natural numbers. N-tuples can be introduced as abbreviations.

$$<a_1,a_2,\ldots,a_n> \equiv <a_1,<a_2,\ldots,a_n>>$$

The sequence of symbols at the left abbreviates the sequence at the right. The symbol $\equiv$ expresses the identity of terms taken as sequences of symbols. R-Maple programs are composed of n-tuples consisting of an operator and $n-1$ operands. Each operator has a fixed arity:

$$<operator, operand_1, operand_2, \ldots, operand_{n-1}>$$

Although the approach taken here is more formal, the reader will recognize a touch of LISP. Abstract programs express the data they operate on as literals. Literals are terms of arithmetic denoting natural numbers and consist of numerals and pairs. Numerals and literals are defined as smallest classes satisfying formation rules. Formation rules for numerals are:

a) The symbol 0 is a numeral.

b) If $\mathfrak{s}$ is a numeral so is $\mathfrak{s}'$.

Formation rules for literals are:

a) Every numeral is a literal.

b) The symbol $\underline{nil}$ is a literal.

c) If $\mathfrak{s}$ and $\mathfrak{t}$ are literals so is $<\mathfrak{s},\mathfrak{t}>$.

Readers objecting to an apparent confusion between natural numbers and pairs should realize that a sequence of bits in computer memory can be also *interpreted* by a program as an integer or a floating-point number. Natural numbers can be viewed either as themselves or as pairs. The intended use must be built into R-Maple programs. R-Maple is a completely typeless language, just as assemblers are. LISP, which is considered a typeless language, has a form of typing called

weak or *dynamic* with checks for type violations performed during the execution.

Another possible objection, that of tricky encoding of pairs into natural numbers will, be dealt with in a moment after a discussion of abstract programs which are encoded in an even "trickier" form. Abstract R-Maple programs are subsets of literals and they consist of two components, control and meaning. A control component is solely concerned with the efficiency of computation. The question of convergence of computations is understood as a question of control. A non-terminating program can be viewed as being computed in an absolutely inefficient way (having the infinite complexity).

Abstract programs are terms of arithmetic and as such they *directly* denote natural numbers in the standard interpretation of arithmetic. These numbers, however, do not express the meaning of programs. The meaning component of an abstract program will be given by a *naming function* denoted by "*" which is defined on the domain of abstract programs (a subset of literals). Its range are terms and formulas of formal arithmetic. R-Maple *expressions*, which are data possibly containing *program* variables, name terms possibly containing *individual* variables. Abstract programs of R-Maple name formulas of arithmetic. The standard interpretation of arithmetic assigns denotations to the terms and formulas. Thus the intended meaning of a program a is the meaning of the formula a^*. The naming function is defined on formal objects (terms and formulas) and thus it belongs to meta-theory rather than to the theory of arithmetic.

Readers familiar with meaning functions employed, in what is called, *the* denotational semantics of programming languages will probably be surprised by the simplicity of the naming function "*". The way meanings are assigned to R-Maple programs can be also called denotational semantics.

Standard denotational semantics assigns meanings into models of lambda calculus as developed by D. Scott [12]. The models of lambda calculus are, obviously, more complicated than models of arithmetic. This additional complexity of lambda calculus models would not normally be an obstacle to *practical* semantics of programming languages. The real obstacle is that there is no workable formal theory of lambda calculus in which properties of programs can be easily derived. We are alluding here to the complications associated with the practical use of Scott's induction rule as opposed to the simplicity of induction in formal arithmetic. Another obstacle is that the meaning functions associated with lambda calculus tend to be quite complicated and generally unworkable.

A program a names the formula a^*. Thus a resembles the Gödel number of the formula a^*. There is a slight difference, however. Not all formulas of arithmetic are in the range of the naming function. Formulas containing universal quantifiers are not named by R-Maple programs. On the other hand, two or more programs having different control components can have the same meaning *trivially* when they synonymously name the same formula. We say trivially because there are R-Maple programs which have identical meanings non-trivially. They name different formulas which can be then *proven* equivalent.

Now we are ready to answer the objection to tricky Gödelization. R-Maple computation, just as computatations in the languages based on lambda calculus, is performed by transformations of terms. The use of "Gödel numbers" in the semantics of R-Maple is just for the *interpretation* of R-Maple programs into the arithmetic. Even so, the interpretation is straight-forward and natural, without any use of prime factorization as perhaps readers might have suspected. R-Maple computation is never hampered by huge Gödel numbers, whereas the semantics of R-Maple provides theorems which - when one is proving properties of R-Maple programs - circumvent the direct use of Gödel numbers.

One of the major advantages of abstract programs of R-Maple and LISP is that programs can be manipulated and computed by other programs. A possible mistake of the designers of LISP was that they did not go beyond abstract programs. Practical programs tend to be quite large and the pragmatic aspect of *readability* of programs by humans (rather than by machines) cannot be overlooked by any designer of programming languages.

The aspect of human engineering is taken into consideration in the concrete syntax of R-Maple programs. Abstract syntax of R-Maple is quite simple and can be defined by simple formation rules as used in mathematical logic. Concrete syntax of programming languages is more complex and contains a variety of syntactic classes. It is traditionaly described by formal grammars, most often by a BNF notation. The correspondence between a concrete and abstract syntax (if a language has one at all) is then either defined by a translation scheme, or completely dismissed as trivial. The latter is almost invariably the case in denotational descriptions of existing programming languages.

We do not think that the correspondence is so trivial that it should be left unexplained. A reader who thinks otherwise should inspect the correspondence given in sections (9) to (13) first. On the other hand, the use of syntax translation schemes belongs to the theory of grammars and translations and cannot be directly related to mathematical logic which forms the basis of our discourse. The solution adapted here is quite novel from the point of view of both programming languages and mathematical logic.

We treat a program in a concrete form as a mere *abbreviation* which stands for a term in abstract form. No translation on the level of the theory is necessary since both forms are the same. Abbreviations are widely used in logic but this approach does not seem to be used in programming languages. On the other hand, the concrete syntax is quite complicated and had we tried to handle it in a standard logical way, by informal descriptions in English, we could have run into many problems of ambiguity. The syntax of abbreviations will be described by BNF rules. The abbreviations are then related via *schemas of abbreviations* to the terms abbreviated.

The reader should always bear in mind that R-Maple programs presented in the *concrete* syntax are not sequences of terminal symbols as specified by the BNF productions. Terminal sentences only stand for R-Maple programs in the abstract syntax. The *concrete* syntax of R-Maple programs does not have the kind of *ontological* independence as in other programming languages. Thus the abbreviations provide a true "syntax sugaring" of the language.

The process of elimination of abbreviations in a concrete program in order to obtain the abstract program will be called *compilation* of the program. A compilation can be performed mechanically by another program called a *compiler*.

In order to prove correctness of compilers of ordinary programming languages one has to show that the meanings of both source and target programs are the same. Compilation in R-Maple is not concerned with the meaning at all. One only has to show that the abbreviations are removed correctly. Programs in the concrete syntax have no autonomous meaning.

5. Abstract Syntax of R-Maple.

Operators of abstract syntax are introduced by the abbreviations:

$\underline{varop} \equiv 0$, $\underline{quoteop} \equiv 1$, $\underline{pairop} \equiv 2$, $\underline{predop} \equiv 3$, $\underline{trueop} \equiv 4$,
$\underline{falseop} \equiv 5$, $\underline{attrop} \equiv 6$, $\underline{andsop} \equiv 7$, $\underline{andpop} \equiv 8$, $\underline{orsop} \equiv 9$,
$\underline{orpop} \equiv 10$, $\underline{notop} \equiv 11$, $\underline{ifop} \equiv 12$, $\underline{findop} \equiv 13$, $\underline{splitop} \equiv 14$,
$\underline{defop} \equiv 15$ $\underline{exop} \equiv 16$, $\underline{whereop} \equiv 17$, $\underline{moveop} \equiv 18$

Abstract R-Maple programs form a subset of literals, thus they are terms of arithmetic. The fact that abstract programs are a subset of R-Maple data enables the reflexivity of programs: programs may operate on programs. For the definition of the class of programs we need to define three auxiliary subsets of literals: *program variables*, *program predicates*, and *expressions*.

Formation rules for these classes are given with the help of meta-variables. Meta-variables are *syntactic* variables ranging over terms. The following syntactic variables will be used.

$\mathbf{Num}, \mathbf{Num}_1, \mathbf{Num}_2, \cdots$ to range over numerals
$\mathbf{Lit}, \mathbf{Lit}_1, \mathbf{Lit}_2, \cdots$ to range over literals
$\mathbf{Var}, \mathbf{Var}_1, \mathbf{Var}_2, \cdots$ to range over program variables
$\mathbf{Pred}, \mathbf{Pred}_1, \mathbf{Pred}_2, \cdots$ to range over program predicates
$\mathbf{Expr}, \mathbf{Expr}_1, \mathbf{Expr}_2, \cdots$ to range over expressions

Prog,**Prog**$_1$,**Prog**$_2$, $\cdots$ to range over programs

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. We are stressing this point because in the description of concrete syntax we shall use symbols as **Pred**, **Expr**, $\cdots$ as non-terminals in BNF productions for the concrete grammar.

Program variables, predicates, expressions, and programs are the least sets of terms satisfying their respective formation rules.

Formation rule for program variables:

<*varop*,**Num**> is a program variable.

Formation rule for program predicates:

<*predop*,**Num**> is a program predicate.

Formation rules for expressions:

a) <*quoteop*,**Lit**> and **Var**, are expressions.

b) If **Expr**$_1$ and **Expr**$_2$ are not both quoted then <*pairop*,**Expr**$_1$,**Expr**$_2$> is an expression.

An expression is *quoted* if it is of the form <*quoteop*,**Lit**>. An expression is *consed* if it is of the form <*pairop*,**Expr**$_1$,**Expr**$_2$>.

Formation rules for programs:

a) <*trueop*,*nil*>, <*falseop*,*nil*>, <*attrop*,**Pred**,**Expr**>, <*ifop*,**Prog**$_1$,**Prog**$_2$,**Prog**$_3$>, <*findop*,**Var**,**Prog**>, <*splitop*,**Var**$_1$,**Var**$_2$,**Expr**,**Prog**>, <*orsop*,**Prog**$_1$,**Prog**$_2$>, <*orpop*,**Prog**$_1$,**Prog**$_2$>, <*andsop*,**Prog**$_1$,**Prog**$_2$>, <*andpop*,**Prog**$_1$,**Prog**$_2$>, <*notop*,**Prog**> are programs.

b) <*exop*,**Prog**>, <*whereop*,**Prog**,**Var**,**Expr**>, <*moveop*,**Var**$_1$,**Var**$_2$,**Prog**> are programs.

Operators of group b) are called *processes* or *program-counters*. They mark positions in programs where transformations take place. Processes are created, delayed, and terminated by the computing machine executing R-Maple programs under direction of the control component of programs. This involves the insertion, modification, and removal of program-counters. "Users" writing programs in R-Maple are not allowed to use the formation rule b). Thus processes are created only by the executing machine.

Formation rules give a purely syntactic characterization of programs. There are some additional syntactic constraints on properly formed programs which cannot be expressed by formation rules. For instance, a properly formed program may not contain free (undeclared) program variables. The additional constraints are explained in section (7).

Expressions of R-Maple may contain program variables which will be replaced during a computation by literals. At the same time R-Maple programs may operate on literals of any form. In order to be able to tell whether

<*pairop*,<*varop*,*3*>,<*quoteop*,*nil*>>

is a consed expression still containing a program variable or whether it is just pure data containing the literal <*varop*,*3*> we have to quote literals. But even with literals quoted we still cannot tell whether <*varop*,*6*> is an expression still containing a program variable or it is a pair with number *30*. Thus we have to cons pairs.

The difference between consed and quoted expressions corresponds to the difference between (*cons a* 3) and (*quote* (*a* . 3)) in LISP. The first expression is meant to be evaluated with the value of the variable *a* to produce a pair whereas the latter expression evaluates directly to a pair.

Expressions in LISP are evaluated to constant S-expressions before they can be used as arguments to functions. Expressions in R-Maple are not evaluated; program variables occurring in them are merely replaced. It is possible to "pass" still "unevaluated" expressions to a predicate. This enables a predicate to be evaluated in parallel with programs computing values of program variables contained in expressions.

It is certainly possible to extend LISP with unevaluated expressions in order to permit *lazy evaluations*, but the schemes the present author knows about [2,3] are not very pleasing from the points of view of both syntax and semantics.

Prolog, as a representative of what is called *logic programming*, relies heavily on the use of unevaluated expressions but it lacks the explicit control component which we deem necessary to enable efficient computations.

We shall now specify the objects named by program variables. Let us order all individual variables of arithmetic alphabetically into a sequence:

$$\underline{a},\underline{b},\underline{c},\ldots,\underline{aa},\underline{ab},\underline{ac},\ldots,\underline{a1},\ldots,\underline{a9},\underline{ba},\cdots$$

Let $\mathfrak{a}$ stand for the i-th symbol (counting from 0) in the sequence. We set

$$<\underline{varop},i>^* \equiv \mathfrak{a}$$

In order to establish the naming for program predicates let us alphabetically order the predicate symbols of arithmetic:

$$\underline{A},\underline{B},\underline{C},\ldots,\underline{Aa},\underline{Ab},\underline{Ac},\ldots,\underline{A1},\ldots,\underline{A9},\underline{Ba},\cdots$$

If the i-th symbol, counting from 0, in the sequence is denoted by $\mathfrak{a}$ then we set

$$<\underline{predop},i>^* \equiv \mathfrak{a}$$

The meaning of the terms of arithmetic named by quoted and consed expressions is specified by induction on the structure of expressions:

$$<\underline{quoteop},\mathbf{Lit}>^* \equiv \mathbf{Lit}$$
$$<\underline{pairop},\mathbf{Expr}_1,\mathbf{Expr}_2>^* \equiv <\mathbf{Expr}_1^*,\mathbf{Expr}_2^*>$$

As an example we have:

$$<\underline{pairop},<\underline{varop},\mathcal{S}>,<\underline{quoteop},\underline{nil}>>^* \equiv <<\underline{varop},\mathcal{S}>^*,<\underline{quoteop},\underline{nil}>^*>$$
$$\equiv <\underline{d},\underline{nil}>$$

The formulas named by programs are specified inductively on the structure of programs:

$$<\underline{trueop},\underline{nil}>^* \equiv \forall \underline{x}\ \underline{x}=\underline{x}$$
$$<\underline{falseop},\underline{nil}>^* \equiv \exists \underline{x}\ \underline{x}\neq\underline{x}$$
$$<\underline{attrop},\mathbf{Pred},\mathbf{Expr}>^* \equiv \mathbf{Pred}^*(\mathbf{Expr}^*)$$
$$<\underline{ifop},\mathbf{Prog}_1,\mathbf{Prog}_2,\mathbf{Prog}_3>^* \equiv (\mathbf{Prog}_1^* \,\&\, \mathbf{Prog}_2^*) \vee (\neg\, \mathbf{Prog}_1^* \,\&\, \mathbf{Prog}_3^*)$$
$$<\underline{findop},\mathbf{Var},\mathbf{Prog}>^* \equiv \exists \mathbf{Var}^*\, \mathbf{Prog}^*$$
$$<\underline{splitop},\mathbf{Var}_1,\mathbf{Var}_2,\mathbf{Expr},\mathbf{Prog}>^* \equiv$$
$$\exists \mathbf{Var}_1^* \exists \mathbf{Var}_2^* (<\mathbf{Var}_1^*,\mathbf{Var}_2^*> = \mathbf{Expr}^* \,\&\, \mathbf{Prog}^*)$$
$$<\underline{orsop},\mathbf{Prog}_1,\mathbf{Prog}_2>^* \equiv \mathbf{Prog}_1^* \vee \mathbf{Prog}_2^*$$
$$<\underline{orpop},\mathbf{Prog}_1,\mathbf{Prog}_2>^* \equiv \mathbf{Prog}_1^* \vee \mathbf{Prog}_2^*$$
$$<\underline{andsop},\mathbf{Prog}_1,\mathbf{Prog}_2>^* \equiv \mathbf{Prog}_1^* \,\&\, \mathbf{Prog}_2^*$$
$$<\underline{andpop},\mathbf{Prog}_1,\mathbf{Prog}_2>^* \equiv \mathbf{Prog}_1^* \,\&\, \mathbf{Prog}_2^*$$
$$<\underline{notop},\mathbf{Prog}>^* \equiv \neg\, \mathbf{Prog}^*$$
$$<\underline{exop},\mathbf{Prog}>^* \equiv \mathbf{Prog}^*$$
$$<\underline{whereop},\mathbf{Prog},\mathbf{Var},\mathbf{Expr}>^* \equiv \mathbf{Prog}^* \,\&\, \mathbf{Var}^* = \mathbf{Expr}^*$$
$$<\underline{moveop},\mathbf{Var}_1,\mathbf{Var}_2,\mathbf{Prog}>^* \equiv \exists \mathbf{Var}_1^*\, \mathbf{Prog}^*$$

The naming function for most of the operators has been already discussed in section (3); the philosophy behind the processes will become obvious once their behavior during a computation will be explained in the sections (15) through (17).

6. Some Syntactic Properties of Programs.

Expressions and programs of R-Maple have similar syntactic properties as terms and formulas of arithmetic. Program variables occurring in terms of R-Maple share syntactic properties with individual variables of arithmetic.

An R-Maple term **a** *contains* the term **b** iff **b** is a subterm of **a** and the occurrence of **b** is not a part of any subterm $<\underline{quoteop},\mathbf{c}>$ of **a**. A program variable *occurs* in a term **a** of R-Maple iff **a** contains the variable. Program variables occurring in programs can be either *free* or *bound*. An occurrence of the program variable **Var** is *free* in the term **a** iff the corresponding occurrence of the individual variable $\mathbf{Var}^*$ is free in the term or formula $\mathbf{a}^*$. An occurrence of the program variable **Var** in the program **Prog** is *bound* iff the corresponding occurrence of the individual variable $\mathbf{Var}^*$ in the formula $\mathbf{Prog}^*$ is bound. Note that program variables cannot be bound in expressions.

Operators $\underline{findop}$ and $\underline{splitop}$ are *variable binding operators*. Both variables in $\underline{splitop}$ are bound. The variable $\mathbf{Var}_1$ is, obviously, bound in $<\underline{moveop},\mathbf{Var}_1,\mathbf{Var}_2,\mathbf{Prog}>$. There is no corresponding variable in the formula of arithmetic for $\mathbf{Var}_2$. We stipulate that $\mathbf{Var}_2$ is free, i.e. that $\underline{moveop}$ binds only its first variable. Program variables bound by these operators are said to be *declared* in the scope of operators.

We shall now define a meta-theoretic function designating terms of R-Maple obtained by *substitution* for program variables in another terms. The substitution for program variables is once again closely correlated to the substitution for individual variables in arithmetic. We shall designate by $\mathbf{a}\{\mathbf{Var}:=\mathbf{Expr}\}$ the term of R-Maple which is obtained by the substitution of the expression **Expr** for all free occurrences of the program variable **Var** in the term **a**. The substitution in expressions, i.e. terms and formulas, of arithmetic is designated by $\mathbf{a}[\mathbf{x}:=\mathbf{s}]$. The substitution for program variables will be defined in such a way that

$$\mathbf{a}\{\mathbf{Var}:=\mathbf{Expr}\}^* \equiv \mathbf{a}^*[\mathbf{Var}^*:=\mathbf{Expr}^*] \tag{1}$$

The substitution for individual variables, as defined in [6,14], is meaningless when a term containing a free variable comes into the range of a variable binding operator with the same variable. It is only cumbersome in logic to make sure that this does not happen; in R-Maple we cannot properly define computations with such an understanding of substitution.

Various devices have been designed to assure that a substitution is always well defined. One widely used device uses different symbols for bound and free individual variables. Thus a free variable of a term can never enter a scope of a bound variable. The problem with this device is that it does not directly permit the applications of the Equivalence Theorem in the scope of quantifiers.

In order to demonstrate the problem let us assume that we have as a theorem

$$\underline{P}(\underline{a}) \leftrightarrow \exists \underline{x} \underline{R}(\underline{a},\underline{x}) \tag{2}$$

It is impossible to replace $\underline{P}(\underline{x})$ by the Equivalence Theorem in the formula

$$\exists \underline{x}(\underline{P}(\underline{x}) \;\&\; \underline{Q}(\underline{x}))$$

because it involves the substitution of bound variable $\underline{x}$ for free variable $\underline{a}$ in the formula (2). Note that had this been allowed, a bound variable would have come into a scope of the same bound variable.

We can, however, use the device of Curry developed for the substitution in lambda calculus [see for instance 5]. We shall use it for both kinds of substitution. A precise meta-theoretic definition of substitution requires an inductive definition on the structure of formulas and terms. Since the definition is otherwise straight-forward, we present here only its crucial case. Substitution involving a variable binding operator, for instance "$\exists$" is defined as follows.

$$(\exists \mathbf{v}\mathbf{A})[\mathbf{w}:=\mathbf{s}] \equiv \begin{cases} \exists \mathbf{v}\mathbf{A} & \text{if } \mathbf{v} \equiv \mathbf{w} \\ \exists \mathbf{v}(\mathbf{A}[\mathbf{w}:=\mathbf{s}]) & \text{if } \neg\, \mathbf{v} \equiv \mathbf{w} \;\&\; \mathbf{v} \text{ is not free in } \mathbf{s} \\ \exists \mathbf{x}(\mathbf{A}[\mathbf{v}:=\mathbf{x}][\mathbf{w}:=\mathbf{s}] & \text{otherwise} \end{cases}$$

where **x** is the first variable in the alphabetic sequence different from **w** not occurring in either **s** or **A**. Bound variable **v** is in the third case first *renamed* by the variable **x**.

Substitution for program variables is defined similarly. As an example let us assume that **Var** denotes the variable with the least index not occurring in either of $\mathbf{Var_2}$, **Prog**, **Expr**; and that $\mathbf{Var_1}$ is free in **Expr**. Then we shall have:

$$<\underline{findop},\mathbf{Var_1},\mathbf{Prog}>\{\mathbf{Var_2}:=\mathbf{Expr}\} \equiv$$
$$<\underline{findop},\mathbf{Var},\mathbf{Prog}\{\mathbf{Var_1}:=\mathbf{Var}\}\{\mathbf{Var_2}:=\mathbf{Expr}\}>$$

In the case of

$$<\underline{splitop}\,\mathbf{Var_1},\mathbf{Var_2},\mathbf{Expr},\mathbf{Prog}>\{\mathbf{Var_3}:=\mathbf{Expr_1}\}$$

variables $\mathbf{Var_1}$ and/or $\mathbf{Var_2}$ must be renamed when free in $\mathbf{Expr_1}$.

There are no bound program variables in expressions. Thus the substitution for a program variable simply replaces all occurrences of the variable by the expression being substituted. An expression previously containing variables can be turned by a substitution into a constant expression. The formation rule b) for expressions (see section (5)) may become violated in the process. The operation of substitution is defined in a such way that it replaces every occurrence of the term

$$<\underline{pairop},<\underline{quoteop},\mathbf{Lit_1}>,<\underline{quoteop},\mathbf{Lit_2}>>$$

which is not an R-Maple expression by

$$<\underline{quoteop},<\mathbf{Lit_1},\mathbf{Lit_2}>>$$

For instance:

$$<\underline{attrop},<\underline{predop},0>,<\underline{pairop},<\underline{pairop},<\underline{varop},1>,<\underline{quoteop},6>>,<\underline{quoteop},\underline{nil}>>$$
$$\{<\underline{varop},1>:=<\underline{quoteop},3>\} \equiv$$
$$<\underline{attrop},<\underline{predop},0>,<\underline{quoteop},<<3,6>,\underline{nil}>$$

The meaning component of expressions is not affected by such substitutions and the property (1) is upheld. In our example we have:

$$<\underline{attrop},<\underline{predop},0,<\underline{quoteop},<<3,6>,\underline{nil}>>^* \equiv \underline{A}(<<\underline{b},6>,\underline{nil}>)[\underline{b}:=3\,]$$

The reader interested in the exact meta-theoretic definition can infer it from its arithmetic, i.e. formal theoretic, counterpart as discussed in Part II.

7. Definitions of Predicates.

Definitions of predicates give names to programs. Named programs turn into predicates. Program predicates are one place (one argument). The effect of many-place predicates is achieved via n-tuples. There are two kinds of predicates tests and generators. Arguments of tests are *input* arguments. Arguments of of generators are either *output* only, or they are pairs of input and output arguments.

Predicates are *invoked* by the operation of attribution: $<\underline{attrop},\mathbf{Pred},\mathbf{Expr}>$. If **Pred** is a test then the evaluation of the atrribution is expected to reduce (to transform) to either **T** or **F**. On the other hand, if **Pred** is, say with both input and output variables, then the argument **Expr** should have the form $[\mathbf{Expr_1},\mathbf{Var}]$ where $\mathbf{Expr_1}$ is the input argument and **Var** is the output variable. The computation of this invocation should result in an assignment of an expression to the output variable **Var**. If there are more possible output values then *backtracking* will be invoked to try as many output values as necessary. If no output value can be found then the invocation should reduce to **F**. The computation, i.e. evaluation, of attributions will be discussed in section (16).

Definitions of predicates have the abstract form:

$$<\underline{defop},\mathbf{Pred},\mathbf{Var},\mathbf{Prog}> \qquad (1)$$

The distinction between tests an generators is on the level of **Prog** in the way it reduces to a truth value or to an assignment. A generator **Pred** is said to be with *input* when it is defined as

<*defop*,Pred,Var,<*splitop*,$\mathbf{Var}_1$,$\mathbf{Var}_2$,Var,Prog>>

$\mathbf{Var}_1$ is the *input* variable, $\mathbf{Var}_2$ is the *output* variable. If the definition (1) of a generator is not of this form then the generator is said to be with *output only*. The variable **Var** is then the output variable.

The distinction between tests, generators, input, and output variables is significant only on the level of control. The intended meaning of definitions is set by:

<*defop*,Pred,Var,Prog>* ≡ ∀Var*(Pred*(Var*) ↔ Prog*)

The intuition behind the intended meaning of definitions is that by writing down a predicate definition one should be able to introduce by the way of recursive extensions of arithmetic a predicate satisfying the intended meaning. This is, however, achievable only if the computation of the predicate can be shown to terminate (see Part II).

An example of a program predicate for which the intended meaning cannot be asserted is the test defined as

<*defop*,<*predop*,*1*>,<*varop*,*0*>,*notop*,*attrop*,<*predop*,*1*>,*varop*,*0*>

The intended meaning

$\forall a(\ \underline{B}(a) \leftrightarrow \neg\ \underline{B}(a)\ )$

can be derived only at the cost of inconsistency.

Definitions of predicates can be grouped together into a list of definitions. Let us use the syntactic variables **Def** and **Defs** to range over predicate definitions and lists of predicate definitions respectively. Lists of definitions are then the least set of terms satisfying the following formation rule.

nil and **<Def,Defs>** are lists of definitions.

The formula named by a list of definitions is the conjunction of formulas named by single definitions:

nil* ≡ $\forall x\ x=x$
<Def,Defs>* ≡ Def* & Defs*

There are some additional constraints on terms used in definition of predicates. These cannot be given by "context-free" formation rules. The constraints correspond to usual constraints of programming languages: all identifiers must be declared; procedures should be invoked with the correct number and kind (input - output) of arguments.

Such constraints are sometimes referred to as semantic constraints, although a more appropriate term is being increasingly used: *context-sensitive syntax*. The constraints on the form, i.e. on the syntax, of terms constituing R-Maple programs are defined in the meta-theory and in English. Formal definition of these constraints is given in Part II.

The program **Prog** of (1) and (2) is called the *body* of the predicate **Pred**; the variable **Var** is called the (*formal*) *argument*. With the exception of formal arguments all program variables occurring in bodies of predicates must be declared (bound).

Predicate **Pred** is *invoked* in a program **Prog** if **Prog** contains a term **<*attrop*,Pred,Expr>**. A predicate **Pred** is invoked in the list of predicate definitions **Defs** if it is invoked in a body of a definition contained in the list. The predicate **Pred** is said to be *predefined* if **Pred*** is one of the following:

Eq,*Ne*,*Lt*,*Le*,*Gt*,*Ge*,*Print*,*Return*,*Add*,*Sub*,*Mul*,*Div*,*Rem*

Predefined predicates (see (14)) do not have explicit definitions. Invocations of predefined predicates are computed in a different way than invocations of defined predicates (see (16)).

Predefined predicates may not be defined in a list of predicate definitions **Defs**. Every other predicate **Pred** invoked in **Defs** must be defined exactly once in **Defs**. There are otherwise no restrictions on simple or mutually recursive invocations of predicates in **Defs**.

Every invocation of the generator **Pred** which is without input must be of the form

$<\underline{attrop}$,**Pred**,**Var**$>$

Every invocation of the generator with input **Pred** must be of the form

$<\underline{attrop}$,**Pred**,$\underline{pairop}$,**Expr**,**Var**$>$

8. Concrete Syntax of Numerals, Constants, Variables, and Predicates.

Concrete syntax of R-Maple is described by BNF productions. Terminal sequences produced from non-terminals are not independent entities but rather abbreviations for terms of abstract R-Maple. The correlation between an abbreviation and a term of abstract R-Maple is given in an informal way for numerals, constants, program variables, and program predicates. The correlation for all other syntactic structures is given by *schemas of abbreviations*.

Abbreviations do not give any *meaning* to syntactic constructs. Obviously, if two different R-Maple constructs a and b abbreviate the same term ($a \equiv b$) then, because of the reflexivity of the identity relation, they *denote* the same object ($a = b$).

BNF productions have the usual form

S ::= $a \mid b \mid c \cdots$

Non-terminals will be capitalized and bold-faced. The symbols **T** and **F** are the only capitalized bold face terminals. Two terminal symbols "|" and "||" are composed of symbols for alternatives. These four terminals will be surrounded by quotes in BNF productions. Everything else on the RHS of productions are terminals.

Numerals and constants abbreviate some of the quoted expressions of abstract syntax. The BNF rules for numerals are as follows.

Num ::= **Digit** | **Num Digit**
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note that "concrete" numerals are in roman font. The abbreviations are given as

$0 \equiv <\underline{quoteop}, 0>$
$1 \equiv <\underline{quoteop}, 1>$
$2 \equiv <\underline{quoteop}, 2>$
$3 \equiv <\underline{quoteop}, 3>$
...

Thus we have a natural correlation between numerals of concrete syntax and arithmetic, for instance: $5^* \equiv 5$.

We follow the standard practice of programming languages and use identifiers for three different lexicographic groups: reserved words, constants, and program variables.

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

Reserved words are the following ones.

div, eq, else, ge, gt, find, if, in, le lt, move, ne, not, or, orp, rem, then, where

In order to increase the readability of programs the reserved words will be printed in boldface.

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

Const ::= *nil*

The abbreviation is:

$nil \equiv <\underline{quoteop}, nil>$

Additional constants can be defined by an *extension* of concrete syntax. Extensions add new BNF productions to the concrete language. Constants are introduced in groups. Each group is given a name. For instance a group of constants $\mathbf{a}_0,\mathbf{a}_1,\mathbf{a}_2,\ \ldots\ ,\mathbf{a}_n$ named **A** can be introduced by a new alternative for **Const**:

Const ::= **A**
A ::= $\mathbf{a}_0 \mid \mathbf{a}_1 \mid \cdots \mid \mathbf{a}_n$

A group of constants is always set to abbreviate numerals 0 through n, i.e. for all $0 \leq i \leq n$:

$\mathbf{a}_i \equiv \langle \underline{quoteop}, i \rangle$

Groups of constants are similar to enumerated types of Pascal.

Let us introduce a group of constants **Opcodes** naming the primitive operators of R-Maple:

Const ::= **Opcodes**
Opcodes ::= *varop* | *quoteop* | *pairop* | *predop* | *trueop* | *falseop* | *attrop* | *andsop* | *andpop* | *orsop* | *orpop* | *notop* | *ifop* | *findop* | *splitop* | *deftop* | *defgop* | *exop* | *whereop* | *moveop*

All constants introduced sofar name the correponding constants of arithmetic, for instance : $nil^* \equiv \underline{nil}$, $ifop^* \equiv \underline{ifop}$.

Identifiers which are neither reserved words nor constants can be used as program variables.

Var ::= **Ident**

The correlation of abbreviations for program variables is obtained in a similar way as their names. Let us order all identifiers alphabetically into the sequence:

$a, b, c, \ldots, aa, ab, ac, \ldots, a1, \ldots, a9, ba, \cdots$

Let **a** stand for the i-th identifier in the sequence (counting from 0). If **a** is neither a reserved word nor a constant then we set

$\mathbf{a} \equiv \langle \underline{varop}, i \rangle$

This correlation gives us the natural property that a program variable names the individual variable with the same identifier. For instance: $result^* \equiv \underline{result}$. On the other hand, not all abstract variables have abbreviations in the concrete syntax.

BNF productions for program predicates are as follows.

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

If **a** is the i-th program predicate in the alphabetic sequence of predicates (**Pred**) and **a** is neither **T** nor **F** then

$\mathbf{a} \equiv \langle \underline{predop}, i \rangle$

We have the natural correlation again: $Add^* \equiv \underline{Add}$.

Program variables, predicates and numerals will be used more often than individual variables, predicate symbols and numerals of arithmetic. This explains why the symbols of arithmetic are underlined.

9. Concrete Syntax of Expressions.

We shall now give a complete list of BNF productions for expressions but will defer the discussion of some constructs until other relevant constructs have been introduced.

Expr ::= **Mexpr** | **Mexpr** + **Expr** | **Mexpr** - **Expr**
Mexpr ::= **Sexpr** | **Sexpr** × **Mexpr** | **Sexpr div Mexpr** | **Sexpr rem Mexpr**
Sexpr ::= **Var** | **Num** | **Const** | | **Expr** , **Exprs** | |
Const (**Exprs**) | **Descr** | (**Expr**)

Descr ::= . **Pred** | . **Pred** (**Exprs**)
Exprs ::= **Expr** | **Expr** , **Exprs**

Terms abbreviated by arithmetic operators "+ ,-,×, **dlv** , **rem** " as well as by "nested" invocations of predicates produced by **Descr** are defined by *contextual abbreviations*. Contextual abbreviations eliminate nested invocations whenever they occur in the context of programs. The discussion of contextual abbreviations must be thus deferred until the abbreviations for programs have been introduced (10).

We shall now give the correlation between concrete and abstract expressions as long as the concrete expressions are produced only by the first alternatives of **Expr** and **Mexpr** and by the first five alternatives of **Sexpr**. Parentheses "( )" are used just for grouping and disappear completely in the abstract syntax.

The abbreviations for **Var** and **Num** and **Cons** have already been discussed.

The abbreviations for consed pairs and consed n-tuples are specified by *schemas of abbreviations*:

$[<\underline{quoteop},\mathbf{Llt}_1>,<\underline{quoteop}\,\mathbf{Llt}_2>] \equiv <\underline{quoteop},<\mathbf{Llt}_1,\mathbf{Llt}_2>>$
$[\mathbf{Expr}_1,\mathbf{Expr}_2] \equiv <\underline{pairop},\mathbf{Expr}_1,\mathbf{Expr}_2>$ if $\mathbf{Expr}_1$ and $\mathbf{Expr}_2$ are not both quoted
$[\mathbf{Expr}_1,\mathbf{Expr}_2,\mathbf{Exprs}] \equiv [\mathbf{Expr}_1,[\mathbf{Expr}_2,\mathbf{Exprs}]]$

Schemas of abbreviations escape the informal use of dots and English in logic when specifying abbreviations. A schema of abbreviations usually stands for an infinite number of abbreviations. It contains syntactic variables. Two or more occurences of the same syntactic variable in a schema always stand for the same term assumed by the syntactic variable. We shall employ symbols for non-terminals, possibly subscripted, as syntactic variables ranging over terminal productions obtained from the corresponding non-terminals.

For example let us find the term abbreviated by the quadruple $[6,c,9,nil]$. We set $\mathbf{Expr}_1 \equiv 6$, $\mathbf{Expr}_2 \equiv c$, and $\mathbf{Exprs} \equiv 9,nil$ in order to obtain $[6,[c,9,nil]]$ by the application of the third schema. Applying the third schema to the second component again we obtain

$[6,c,9,nil] \equiv [6,[c,[9,nil]]]$

Now all terminal productions of **Exprs** have been eliminated and the quadruple has been reduced to consed pairs. Using the abbreviations for *nil* and numerals while applying the first schema we have:

$[9,nil] \equiv [<\underline{quoteop},9>,<\underline{quoteop},\underline{nil}>] \equiv <\underline{quoteop},<9,\underline{nil}>>$

Since $c \equiv <\underline{varop},2>$ is not quoted, the second schema applies in:

$[c,[9,nil]] \equiv [<\underline{varop},2>,<\underline{quoteop},9,\underline{nil}>] \equiv <\underline{pairop},<\underline{varop},2>,\underline{quoteop},9,\underline{nil}>$

This is consed, and thus not quoted, and the second schema applies again:

$[6,c,9,nil] \equiv <\underline{pairop},<\underline{quoteop},6>,\underline{pairop},<\underline{varop},2>,\underline{quoteop},9,\underline{nil}>$

Note how the abbreviations for consed pairs take into account the formation rule b) of abstract expressions (section (5)) and at the same time permit natural notation for constants of R-Maple. The meaning function maps expressions of R-Maple in a straight-forward way to the terms of arithmetic:

$[[6,2,nil],[a,6],b]^* \equiv <<6,2,\underline{nil}>,<\underline{a},6>,\underline{b}>$

We also have the "natural" property of substitution:

$[\mathbf{Expr}_1,\mathbf{Expr}_2]\{\mathbf{Var}:=\mathbf{Expr}_3\} \equiv [\mathbf{Expr}_1\{\mathbf{Var}:=\mathbf{Expr}_3\},\mathbf{Expr}_2\{\mathbf{Var}:=\mathbf{Expr}_3\}]$

Since the abbreviations and substitutions automatically quote literals one can naturally write literals in a fully consed notation:

$[[pairop,6,8,],a] \equiv <\underline{pairop},<\underline{quoteop},\underline{pairop},6,8>,\underline{varop},0>$

One can do the same thing in LISP but not in a readable way since LISP does not have concrete

syntax. Moreover, the consed notation in LISP means evaluation.

The abbreviations for the fifth alternative of **Sexpr** are

Const(Exprs) ≡ **[Const,Exprs]**

Thus we have a neat notation for expressions similar to the *terms* of Prolog. For instance

attrop(*predop*(2),*pairop*(*varop*(6),7)) ≡ [*attrop*,[*predop*,2],[*pairop*,[*varop*,6],7]]

10. Concrete Syntax of Programs.

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

Prog ::= **Dprog** |
if Prog then Prog else Prog |
find Decls in Prog |
move Var beyond Var in Prog
Dprog ::= **Cprog** | **Cprog or Dprog** | **Cprog orp Dprog**
Cprog ::= **Sprog** | **Sprog ; Cprog** | **Sprog " || " Cprog** |
Sprog where Var := Expr
Sprog ::= **" T "** | **" F "** | **not Sprog** | **Sprog !** | **Atomp** | **Var := Expr** | **(Prog)**
Atomp ::= **Invoc** | **Expr Relop Expr**
Relop ::= **eq** | **ne** | **lt** | **le** | **gt** | **ge**
Invoc ::= **Pred (Exprs)** | **Pred(Exprsopt " | " Var)**
Exprsopt ::= **Exprs** |
Decls ::= **Decl** | **Decl ; Decls**
Decl ::= **Var** | **Svar := Expr** | **Pred(Exprsopt " | " Svars)**
Svar ::= **Var** | **[Svar , Svars]**
Svars ::= **Svar** | **Svar , Svars**

Schemas of abbreviations will be given in the reverse order of BNF productions: simple constructs first.

Predicates of R-Maple are single argument. Invocations with more arguments are reduced to invocations with single arguments:

Pred(Expr) ≡ **<** *attrop*,**Pred,Expr>**
Pred(Expr,Exprs) ≡ **Pred([Expr,Exprs])**

These schemas are used for invocations of tests.

Invocations of generators place the output variable in a distinguished syntactic position.

Pred(| Var) ≡ **Pred(Var)**

This schema is for generators without input.

Generators with input are invoked with the help of two schemas:

Pred(Expr | Var) ≡ **Pred([Expr,Var])**
Pred(Expr,Exprs | Var) ≡ **Pred([Expr,Exprs] | Var)**

Assignments are reduced to invocations of predefined generator *Return* [see (14)]:

Var:=Expr ≡ *Return*(**Expr | Var**)

The term *assignment* is used only in a figurative sense. R-Maple is an "applicative" language without any notion of states.

Comparisons are reduced to invocations of predefined comparison tests [see (14)]:

$\mathbf{Expr_1}$ **eq** $\mathbf{Expr_2}$ ≡ *Eq*(**Expr**$_1$,**Expr**$_2$)
$\mathbf{Expr_1}$ **ne** $\mathbf{Expr_2}$ ≡ *Ne*(**Expr**$_1$,**Expr**$_2$)
$\mathbf{Expr_1}$ **lt** $\mathbf{Expr_2}$ ≡ *Lt*(**Expr**$_1$,**Expr**$_2$)

Expr$_1$ le Expr$_2$ ≡ *Le*(**Expr$_1$,Expr$_2$**)
Expr$_1$ gt Expr$_2$ ≡ *Gt*(**Expr$_1$,Expr$_2$**)
Expr$_1$ ge Expr$_2$ ≡ *Ge*(**Expr$_1$,Expr$_2$**)

The remaining simple programs have straight-forward abbreviations:

T ≡ <*trueop*,*nil*>
F ≡ <*falseop*,*nil*>
not Simple ≡ <*notop*,**Simple**>
Simple! ≡ <*exop*,**Simple**>

Abbreviations for conjunctions:

Sprog; Cprog ≡ <*andsop*,**Sprog,Cprog**>
Sprog || Cprog ≡ <*andpop*,**Sprog,Cprog**>
Sprog where Var:=Expr ≡ <*whereop*,**Sprog,Var,Expr**>

Abbreviations for disjunctions:

Cprog or Dprog ≡ <*orsop*,**Cprog,Dprog**>
Cprog orp Dprog ≡ <*orpop*,**Cprog,Dprog**>

Abbreviations for decisions and moves are straight-forward:

if Prog$_1$ then Prog$_2$ else Prog$_3$ ≡ <*ifop*,**Prog$_1$,Prog$_2$,Prog$_3$**>
move Var$_1$ beyond Var$_2$ in Prog ≡ <*moveop*,**Var$_1$,Var$_2$,Prog**>

Searches permit us to specify output variables and optionally generators supplying values for the variables. Variables and/or generators are specified in **Decls** and they are used in **Prog**. At the same time it is possible to break down a complex n-tuple into components and assign them to separate program variables.

Declarations with syntax and semantics somewhat similar to our searches are often used in functional programming languages. However, there is an important difference: R-Maple works with relations rather than functions and it may happen that values generated in **Decls** do not satisfy **Prog** and new values must be produced by *backtracking* into the generators [see (17)]. There is no backtracking in declarations of functional programming languages since functions produce unique values.

We shall give the abbreviations for searches in a sequence which demonstrates how complex searches can be reduced to simpler ones by the elimination of abbreviations. The order of removal of abbreviations is actually immaterial since the schemas for searches can be applied in a unique way only.

Multiple searches are reduced to simple searches by:

find Decl; Decls in Prog ≡ **find Decl in find Decls in Prog**

The first alternative of **Decl** directly abbreviates a basic operator:

find Var in Prog ≡ <*findop*,**Var,Prog**>

This is the *unbounded* form of a search. The remaining two alternatives of **Decl** specify a *bounded* search with an explicit generator producing, or *declaring*, the value searched for.

The second alternative of **Decl** is reduced to the the third form by:

find Svar:=Expr in Prog ≡ **find** *Return*(**Expr | Svar**) **in Prog**

Bounded searches with simple variables are unbounded searches written in a shorter notation:

find Pred(Exprsopt | Var) in Prog ≡ **find Var in Pred(Exprsopt | Var); (Prog)**

Searches permit to break the single value of the output variable produced by a generator into its constituents when the value is an n-tuple. The constituents are then assigned to simple variables from a list of structured variables.

Searches with lists of structured output variables are first reduced to searches with structured output variables:

find Pred(Exprsopt | Svar,Svars) in Prog ≡
** find Pred(Exprsopt | [Svar,Svars]) in Prog**

There are four different forms of structured variables:

[[Svar,Svars$_1$],Svars$_2$], [Var,Svar,Svars], [Var,[Svar,Svars]], [Var$_1$,Var$_2$]

The schemas of abbreviations given below will reduce forms occurring earlier in the list to simpler forms later in the list.

Abbreviations for structured output variables with first components being again structured variables are removed first:

find Pred(Exprsopt | [[Svar,Svars$_1$],Svars$_2$]) in Prog ≡
** find Pred(Exprsopt | [Var,Svars$_2$]); [Svar,Svars$_1$]:=Var in Prog**

This schema of abbreviations introduces (or eliminates) a new bound program variable **Var** which is not present in the abbreviated form.

Similar introduction of bound variables by abbreviations occurs in logic when, for instance, one defines the predicate "$\leq$" by the abbreviation

$s \leq t \equiv \exists_x\, s + x = t$

Intuitively, it does not matter which bound variable is introduced when the abbreviation is removed as long as the new variable is not free in the terms **s** and **t**. Technically, however, one has to present a definite construct which the abbreviation stands for. Let us therefore stipulate that the syntactic variable **Var** stands for the program variable $\langle \underline{varop}, i \rangle$ with the least index i which does not occur in any of the terms within the abbreviation schema.

From now on, whenever an abbreviation schema introduces a new bound variable we shall tacitly assume that the introduced variable is the first one not occuring in the schema.

The second form of structured variables is reduced to the third one by:

find Pred(Exprsopt | [Var,Svar,Svars]) in Prog ≡
** find Pred(Exprsopt | [Var,[Svar,Svars]]) in Prog**

The structured variable as the second component of a structured variable (first component being a simple variable) is removed by:

find Pred(Exprsopt | [Var$_1$,[Svar,Svars]]) in Prog ≡
** find Pred(Exprsopt | [Var$_1$,Var]); [Svar,Svars]:=Var in Prog**

The fourth form of structured output variables reduces differently when **Pred** is not *Return*:

find Pred(Exprsopt | [Var$_1$,Var$_2$]) in Prog ≡
** find Pred(Exprsopt | Var); [Var$_1$,Var$_2$]:=Var in Prog**
** if ¬ Pred ≡ Return**

It remains to tackle the case of **Pred** standing for *Return*.

R-Maple does not contain primitive projection functions. The only way to access components of pairs is through *split* searches abbreviating directly a basic operator:

find *Return*(Expr | [Var$_1$,Var$_2$]) in Prog ≡ $\langle \underline{splitop}$,Var$_1$,Var$_2$,Expr,Prog$\rangle$

Note that the typical search

find [*hd*,*tl*]:=*list* **in** $P(tl,hd)$

abbreviates by the above schemas directly to the primitive operation:

$\langle \underline{splitop}, hd, tl, list, P(tl,hd) \rangle$

Examples of searches will be given in next section.

11. Concrete Syntax of Definitions of Predicates

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

Def ::= Pred (Svars) is Prog |
Pred (Svarsopt "|" Var) is Prog |
Clauses
Svarsopt ::= Svars |

This first alternative is used to define tests. Generators are defined by the second alternative. Both forms of definitions are called *procedural* definitions. This is because they resemble the usual definitions of procedures. The third alternative permits the definitions of predicates in a *clausal form* similar to Prolog. Clauses will be discussed in the next section.

Abbreviations for definitions of tests:

Pred(Var) is Prog ≡ <defop,Pred,Var,Prog>
Pred([Svar,Svars]) is Prog ≡ Pred(Var) is find [Svar,Svars]:=Var in Prog
Pred(Svar,Svars) is Prog ≡ Pred([Svar,Svars]) is Prog

Abbreviations for definitions of generators without input:

Pred(| Var) is Prog ≡ Pred(Var) is Prog

Abbreviations for definitions of generators with input:

Pred(Svar | Var) is Prog ≡ Pred(Svar,Var) is Prog
Pred(Svar,Svars | Var) is Prog ≡ Pred([Svar,Svars] | Var) is Prog

The generator *Append*(*frs*,*scnd* | *res*) with two inputs which appends the list *scnd* to the list *frs* to form the output list *res* can be defined as:

Append(*frs*,*scnd* | *res*) **is**
if *frs* **eq** *nil* **then** *res*:=*scnd*
else find [*hd*,*tl*]:=*frs*; *Append*(*tl*,*scnd* | *ares*) **in** *res*:=[*hd*,*ares*]

This form of *Append* is not a final one. Additional syntax sugaring eliminating the auxiliary search for *ares* will be given in the next section.

As an illustration we remove from the above definition of *Append* all abbreviations except the ones directly related to abstract operators.

Append(*a*) **is**
find [*b*,*res*]:=*a* **in**
find [*frs*,*scnd*]:=*b* **in**
if *frs* **eq** *nil* **then** *Return*([*scnd*,*res*])
else find [*hd*,*tl*]:=*frs* **in find** *ares* **in**
Append([[*tl*,*scnd*],*ares*]); *Return*([[*hd*,*ares*],*res*])

Let us give an example of a generator with structured output. Consider the generator *Split*(*lst* | *lst*2) which is the contraposition of *Append*. *Split*(*lst* | *lst*2) generates all possible *pairs* of lists *lst*2 which can be transformed back to *lst* by *Append*(*lst*2 | *lst*).

The definition of *Split* without syntax sugaring for multiple results is :

Split(*lst* | *lst*2) **is** *lst*2:=[*nil*,*lst*] **or**
(**find** [*hd*,*tl*]:=*lst*; *Split*(*tl* | *alst*2); [*fr*,*sc*]:=*alst*2 **in** *lst*2:=[[*hd*,*fr*],*sc*])

The same definition using the full syntactic power of bounded searches is:

Split(*lst* | *lst*2) **is** *lst*2:=[*nil*,*lst*] **or**
(**find** [*hd*,*tl*]:=*lst*; *Split*(*tl* | *fr*,*sc*) **in** *lst*2:=[[*hd*,*fr*],*sc*])

We shall show in Part II how to introduce predicate symbols *Append* and *Split* into formal arithmetic in such a way that the intended meaning of the above definitions is satisfied. With the predicates introduced one can prove by induction:

Append(<*frs*,*scnd*>,*res*) ↔ *Split*(*res*,<*frs*,*scnd*>)

Readers familiar with Prolog will note that there is no need for *Split* in Prolog. *Append* works both ways.

Is there a reason to distinguish between input and output variables at all? As the meaning function confirms, there is no such need from the point of view of meaning. Arguments of logic programmers that there is no need to distinguish the kinds of variables also from the point of view of control are usually illustrated by such predicates as *Append*.

Append works both ways because it is defined in a simple way over the structure of lists. Such predicates can be computed both ways with reasonable efficiency. Predicates which rely on arithmetic, rather than structural, properties are usually *one way* predicates. Two such predicate *Intree* and *Gentree* relying on a mixture of structural and arithmetic properties will be shown in a Prolog-like notation in section (12). $Intree(n,t)$ tests whether n is in the binary search tree t, $Gentree(t \mid n)$ is the contraposition successively generating into n all elements of t. It will be shown that even in Prolog the programmers would write two different programs although from a logical point of view one, working both ways, suffices.

Another example is the following Prolog predicate.

$$Fac(0,1)$$
$$Fac(n',r) \leftarrow Fac(n,aux),\ r = n' \times aux$$

The only sensible way to use Fac is with the first argument as input and the second one as output. An attempt to use it the other way, as for instance $Fac(x,120)$, leads - on most interpreters of Prolog - to non-terminating computations. An ultra-sophisticated Prolog interpreter would be probably capable of computing the factorial the other way, though at a significant cost of efficiency. But unfortunately, it is in the nature of arithmetic functions that they can be computed faster in the natural way, and subsequently there is no hope for ever designing an interpreter working with reasonable efficiency both ways. The existence of truly one-way functions, as the ones used in the public key cryptography [11], should make this point obvious.

The fact that two semantically similar predicates such as *Append* and *Split* have two quite different R-Maple programs underscores the fact that a practical programming language *cannot afford* to ignore the control component. After all, some Prolog *compilers* expect the input-output indication for the user-defined predicates.

12. Clausal Form of Predicate Definitions.

Procedural definitions of R-Maple predicates in the concrete syntax do not deviate too much from the standard style of definitions in other programming languages. There is a growing community of programmers who prefer the elegant definitions of predicates in Prolog. Clauses of Prolog keep the definitions simple and when read as implications they express directly the properties of predicates.

The predicate Append is defined in Prolog as

$$Append(nil,a,a)$$
$$Append([hd,tl],a,[hd,aux]) \leftarrow Append(tl,a,aux)$$

We shall now give BNF productions and abbreviations for **Clauses** permitting a clausal form of definitions. Clausal definitions introduce the elegance of Prolog syntax without its semantic shortcomings. By shortcomings we mean the cuts.

Clauses ::= **Clause** | **Clause Clauses**
Clause ::= **Invoc** . | **Invoc** ← **Atompsopt** . | **Invoc** ← **Atompsopt** "|" **Progopt** .
Atomps ::= **Atomp** | **Atomps** ; **Atomp**
Atompsopt ::= **Atomps** |
Progopt ::= **Prog** |

A clause of the form **Invoc** ← **Atomps** | **Prog**. has as its meaning the formula

Atomps* & **Prog*** → **Invoc***

If one of the terms in the antecedent is empty then the the formula must be adjusted in the

obvious way. The meaning of **Clauses** is the conjunction of meanings of single clauses. We shall use the term *meaning* informally, instead of the function "*", because the clauses taken separately do not correspond to conctructs of R-Maple.

Schemas of abbreviations for **Clauses** permit us to transform any procedural definition to the clausal form. The transformation is achieved by a sequence of steps which remove from the definition sequential disjunctions, decisions, split searches, identities, and assignments.

For each schema of the form **A** ≡ **B** we have as tautology **B'** → **A'** where **A'**,**B'** are meanings of **A**,**B** respectively. The first abbreviation has as **B** a definition of a predicate (**Def**). It follows by the transitivity of implication that for every clause **C** derivable by a sequence of abbreviations from the predicate definition **B** we have **B*** → **C'**. Thus under the assumption that **B*** can be asserted, the meaning **C'** of every constituent clause can be asserted as the property of the defined predicate.

First abbreviation weakens an equivalence to an implication.

Pred(Svars) ← | **Prog** ≡ **Pred(Svars) is Prog**.
Pred(Svarsopt | Var) ← | **Prog** ≡ **Pred(Svarsopt | Var) is Prog**

The meaning is obviously upheld by the tautology:

$\forall x(P(x) \leftrightarrow \mathbf{A}) \rightarrow (\mathbf{A} \rightarrow P(x))$

A "clause" of the above form has a little resemblance to the clauses of Prolog, but at least we made good on our promise that every definition of a predicate can be transformed into a "clausal" form. The antecedent of the clause has now assumed a form of an invocation. Following schemas of abbreviations will gradually move the marker "|" to the right in the body **Prog**.

An *or-elimination* removes a sequential *or* from a body by creating two clauses:

Invoc ← **Atompsopt** | **Cprog**.
Invoc ← **Atompsopt** | **Dprog**. ≡ **Invoc** ← **Atompsopt** | **Cprog or Dprog**.

Or eliminations rely on the tautology

$$(\mathbf{A} \vee \mathbf{B} \rightarrow \mathbf{C}) \rightarrow (\mathbf{A} \rightarrow \mathbf{C}) \,\&\, (\mathbf{B} \rightarrow \mathbf{C}) \qquad (1)$$

Let us denote by **Prog'** the *opposite* of the program **Prog**. The opposite of **Prog** is generally **not (Prog)** unless **Prog** is an invocation of a comparison (or its abbreviation). In the latter case, the pairs of invocations of *Eq*,*Ne* and *Lt*,*Ge* and *Gt*,*Le* are opposite to each other. For instance

(x **lt** 6)' ≡ x **ge** 6

An *if-elimination* removes a decision from the antecedent of a clause. If-eliminations are based on the intended meaning of **if** and on (1).

Invoc ← **Atompsopt** | **(Prog)**; **(Prog$_1$)**.
Invoc ← **Atompsopt** | **(Prog'**); **(Prog$_2$)**. ≡
  Invoc ← **Atompsopt** | **if Prog then Prog$_1$ else Prog$_2$**.

A *find-elimination* removes a search from the antecedent of a clause.

Invoc ← **Atompsopt** | **Prog**. ≡ **Invoc** ← **Atompsopt** | **find Var in Prog**.
  if **Var** not free in **Invoc** and **Atompsopt**

Find eliminations are based on the tautology

$(\exists x \mathbf{A} \rightarrow \mathbf{B}) \rightarrow (\mathbf{A} \rightarrow \mathbf{B})$ provided x is not free in **B**

Identity and *assignment* eliminations remove identities or assignments from antecedents of clauses.

Invoc{Var:=Expr} ← **Atompsopt{Var:=Expr}** | **Cprog**. ≡
  Invoc ← **Atompsopt** | **Atomp**; **Cprog**.
    if **Atomp** ≡ Eq(**Var**,**Expr**) ∨ **Atomp** ≡ **Var eq Expr** ∨
    Atomp ≡ Return(**Expr** | **Var**) ∨ **Atomp** ≡ **Var:=Expr** &
    Var not free in **Invoc** and **Atompsopt**

These eliminations are based on the tautology

$$(x = \mathbf{s} \rightarrow \mathbf{A}) \rightarrow \mathbf{A}[x{:=}\mathbf{s}] \qquad (2)$$

provided x is not free in **s** or **A**.

And-eliminations move an invocation from one side of the bar to the other; they are justified trivially.

Invoc ← **Atomp** | **Cprog**. ≡ **Invoc** ← | **Atomp**; **Cprog**.
Invoc ← **Atomps**; **Atomp** | **Cprog**. ≡ **Invoc** ← **Atomps** | **Atomp**; **Cprog**.
provided an identity or assignment elimination is not applicable

A *split-elimination* removes a split search from the antecedent of a clause.

Invoc{**Var**:=[**Var**$_1$,**Var**$_2$]} ← **Atompsopt**{**Var**:=[**Var**$_1$,**Var**$_2$]} | **Prog**. ≡
Invoc ← **Atompsopt** | **find** [**Var**$_1$,**Var**$_2$]:=**Var** **in** **Prog**.
if **Var**$_1$,**Var**$_2$ not free in **Invoc** and **Atompsopt**

Split eliminations are a combination of find and identity eliminations. If a split search is used in a context similar to the following one

if x **eq** *nil* **then** **Prog** **else** **find** [y,z]:=x **in** **Prog**$_1$

then the following abbreviations will be applicable in the second clause immediately after the split-elimination.

Invoc ← | **Prog**. ≡ **Invoc** ← [**Var**$_1$,**Var**$_2$] **ne** *nil* | **Prog**.
Invoc ← **Atomps** | **Prog**. ≡ **Invoc** ← **Atomps**; [**Var**$_1$,**Var**$_2$] **ne** *nil* | **Prog**.

The transformations are based on the property (4.2) of pairs.

When the bar in a clause reaches the right end it can be removed by:

Invoc ← **Atomsopt**. ≡ **Invoc** ← **Atomsopt** | .

The above transformations do not correlate all terminal productions of **Clauses** to the abstract syntax. As an context sensitive semantic constraint we stipulate that only **Clauses** abbreviating a predicate definition are considered legal.

The removal of abbreviations for clauses can be seen as a form of *compile-time unifications*. Unifications are used by the interpreters of Prolog in the run-time. Unifications are quite time consuming, thus the compilers of Prolog perform essentially the same transformations as ours to convert into procedural definitions.

The unification in Prolog is a little bit smarter than ours. For instance the variables can be systematically renamed. We did not include renaming because it would needlessly complicate the abbreviations and the gain would be only marginal.

On the other hand, our clauses permit a mixed clausal and procedural form of definitions with the bar not fully removed. The mixed mode allows universal quantifiers in antecedents of clauses (via **not find** $\cdots$). This is, obviously, impossible in Prolog.

It should be noted that our clauses are stricter than the clauses of Prolog. A sequential order is *prescribed* by the use of sequential operators in the clauses. The de facto sequential control of Prolog interpreters exhibits identical behavior.

Let us convert the generator *Append* into clausal form. After the introduction of " ← " and after if-elimination we have two clauses:

Append(*frs*,*scnd* | *res*) ← | *frs* **eq** *nil*; *res*:=*scnd*. ≡ (3)
Append(*nil*,*scnd* | *res*) ← | *res*:=*scnd*. ≡ (4)
Append(*nil*,*scnd* | *scnd*) ← | . ≡ *Append*(*nil*,*scnd* | *scnd*).

Append(*frs*,*scnd* | *res*) ← | *frs* **ne** *nil*;
(**find** [*hd*,*tl*]:=*frs*; *Append*(*tl*,*scnd* | *ares* **in** *res*:=[*hd*,*aresc*]). ≡ (5)
Append([*hd*,*tl*],*scnd* | *res*) ← | *Append*(*tl*,*scnd* | *ares*); *res*:=[*hd*,*ares*]. ≡ (6)
Append([*hd*,*tl*],*scnd* | *res*) ← *Append*(*tl*,*scnd* | *ares*) | *res*:=[*hd*,*ares*]. ≡ (7)

$Append([\,hd,tl\,],scnd \mid hd,ares) \leftarrow Append(tl,scnd \mid ares)$.

The transitions from (3), (4), and (7) are by the elimination of identities and assignments. The step from (5) eliminates the split search and the superfluous invocation of $[\,hd,tl\,]$ **ne** *nil*. And-elimination is applied in the step from (6).

This form of *Append*, which is not yet a final one, is exactly as in Prolog:

$Append(nil,scnd \mid scnd)$.
$Append([\,hd,tl\,],scnd \mid hd,ares) \leftarrow Append(tl,scnd \mid ares)$.

Actually almost all clausal definitions look as in Prolog. The significant difference is that we do not compute with the help of clauses and can afford to use negations in antecedents. Negations appear in antecedents after if-eliminations. Prolog has to use cuts to escape double evaluation of tests.

The clausal form of *Split* is:

$Split(lst \mid nil,lst)$.
$Split(hd,tl \mid [\,hd,fr\,],sc) \leftarrow Split(tl \mid fr,sc)$.

Let as give the clausal definitions of *Intree* and *Gentree*:

$Intree(n,m,l,r) \leftarrow n$ **lt** m; $Intree(n,l)$.
$Intree(n,n,l,r)$.
$Intree(n,m,l,r) \leftarrow n$ **gt** m; $Intree(n,r)$.
$Gentree(m,l,r \mid n) \leftarrow Gentree(l \mid n)$.
$Gentree(n,l,r \mid n)$.
$Gentree(m,l,r \mid n) \leftarrow Gentree(r \mid n)$.

In Prolog one could use *Gentree* also as a test. But such a use would defeat the very purpose of binary search trees where the comparison against the top node saves the search of one side of the tree. We can see that the knowledge that we are going to use *Gentree* only as a test allows an efficient employment of control.

13. Descriptions.

There is one frequently mentioned advantage of functional programming languages over programming languages based on relations. It is the possibility of composing function applications into complex terms. In relational programming languages new auxiliary variables have to introduced to name intermediate results.

Whereas in a functional language one would write

$Print(Append(Append([\,1,nil\,],[\,2,nil\,]),[\,3,nil\,]))$

we have to write

find $Append([\,1,nil\,],[\,2,nil\,] \mid a)$; $Append(a,[\,3,nil\,] \mid b)$ **in** $Print(b)$

Note that the advantage of functional programming languages does not lie in greater computational power. The advantage is purely syntactical: a compact and more readable notation.

The same problem of readability occurs in logic where it is solved either by conservative extensions of theories by introduction of new functions or, alternatively, by the use of descriptions as proposed by B. Russell. Whenever we are able to show that the predicate $\underline{P}(\underline{a},\underline{b})$ satisfies the existence and uniqueness properties:

$\exists \underline{b} \underline{P}(\underline{a},\underline{b})$
$\underline{P}(\underline{a},\underline{b}) \,\&\, \underline{P}(\underline{a},\underline{c}) \rightarrow \underline{b} = \underline{c}$

we are justified to introduce (by a conservative extension) a new function $p(\underline{a})$ by the defining axiom

$p(\underline{a}) = \underline{b} \leftrightarrow \underline{P}(\underline{a},\underline{b})$

As it is suggested by the term *conservative* this new axiom does not lend any more power to

the theory being extended. Whatever can be proven with the axiom can be proven without it after systematic elimination of terms $p(\mathbf{s})$.

Descriptions are of the form $\iota \underline{b} P(\underline{a},\underline{b})$ which is read as "the unique $\underline{b}$ such that $P(\underline{a},\underline{b})$ if there is such". Descriptions are treated as *contextual abbreviations*. Every occurrence of a description in a context of formulas can be eliminated. For instance

$$R(\iota \underline{b} P(\underline{a},\underline{b})) \equiv \exists \underline{b}(\ \forall \underline{c}(P(\underline{a},\underline{c}) \leftrightarrow \underline{c} = \underline{b}) \ \&\ R(\underline{b})\ )$$

Our generator *Append* generates a unique value for all input arguments. Thus we could either introduce a function *append* or resort to descriptions. But functional relations, i.e. relations satisfying the existence and uniqueness properties, are only special - if quite common - cases among relations. R-Maple is based on generators which can produce more than one value. It does not seem to be advantageous to introduce functions to take care only of the special case. Indeterminate descriptions of Hilbert [4] are more suitable. The notation $\epsilon \underline{b} P(\underline{a},\underline{b})$ can be read as "any $\underline{b}$ such that $P(\underline{a},\underline{b})$ if there is such". Now only the existence property has to be satisfied.

There are systems of predicate logic which have Hilbert's ϵ-notation as primitive. We are, however, not prepared to use such expressive logical systems as the basis for R-Maple. We shall view indeterminate descriptions as contextual abbreviations:

$$R(\epsilon \underline{b} P(\underline{a},\underline{b})) \equiv \exists \underline{b}(\ P(\underline{a},\underline{b}) \ \&\ R(\underline{b})\ )$$

Due to distinguished syntactic positions of output variables in invocations of R-Maple generators we can escape the use of explicit variables bound by the operator ϵ. Descriptions of the form $.P(x)$ are used as $\epsilon a P(x \mid a)$. Indeterminate descriptions in a context of programs can be then eliminated, as for instance:

$$z \ \mathbf{eq}\ [6,.P(x)] \equiv \mathbf{find}\ P(x \mid a)\ \mathbf{in}\ z \ \mathbf{eq}\ [6,a]$$

Indeterminate descriptions are obtained by applications of the BNF production **Descr**.

We say that an occurrence of a program variable is in *output position* if the occurence is the output variable of a generator invocation. When an occurrence of the program variable **Var** is in $\mathbf{Expr}_2$ and $\mathbf{Expr}_1$ contains a description then any expression containing $[\mathbf{Expr}_1,\mathbf{Expr}_2]$ is said to contain a description *in front* of the occurrence of **Var**.

Let us assume that the program **Sexpr** is a predicate invocation. Furthermore, assume that **Sexpr** contains *exactly* one occurrence of the variable **Var** which is, however, not in the output position and that no description is in front of it. Also assume that **Exprs** does not contain any descriptions. Then we set:

Sexpr{Var:= .Pred} $\equiv$ **find Pred(| Var) In Sexpr**
Sexpr{Var:= .Pred(Exprs)} $\equiv$ **find Pred(Exprs | Var) In Sexpr**

As an additional constraint we stipulate that, apart of its distinguished position in **Sexpr**, the program variable **Var** does not occur in any other term within the schema and that it is the first in the sequence of variables not occurring in the schema. This requirement assures that when a description is eliminated a new variable with the smallest possible index is introduced.

When descriptions, both in predicate calculus and R-Maple, are treated as contextual abbreviations then the scope to which a description applies, i.e. the scope of the introduced existential quantifier, and the order of elimination of descriptions is significant. Because of the notion of control in R-Maple, the ordering of descriptions is crucial. The scope of descriptions in R-Maple is limited to invocations. The order of elimination is given by the above constraints: the *leftmost* and *innermost* description is removed first. For instance:

$$\begin{aligned}
&.P(x)\ \mathbf{eq}\ [6,.Q(.P(x),y)] \equiv \\
&\quad \mathbf{find}\ P(x \mid a)\ \mathbf{in}\ a\ \mathbf{eq}\ [6,.Q(.P(x),y)] \equiv \\
&\quad \mathbf{find}\ P(x \mid a);\ P(x \mid b)\ \mathbf{in}\ a\ \mathbf{eq}\ [6,.Q(b,y)] \equiv \\
&\quad \mathbf{find}\ P(x \mid a);\ P(x \mid b);\ Q(b,y \mid c)\ \mathbf{in}\ a\ \mathbf{eq}\ [6,c]
\end{aligned}$$

Note that what can be at first sight considered a "common subexpression": $.P(x)$ *must* be transformed into two separate searches because P can be a generator producing multiple results

and two separate backtrackings may be initiated before the identity test is satisfied. Common subexpressions in Algol-60 cannot be optimized by compilers because of possible side effects during the evaluation of two identical function calls. Side effects within expressions in modern programming languages are considered harmuful. We have retained the syntactic mark "." of descriptions to underscore the fact that descriptions are indeterminate and generally do not stand for the same value.

Computation of nested descriptions corresponds to what is called a *normal order* of evaluation: evaluate all arguments from left to right before applying a procedure.

Readers familiar with more relaxed constraints on evaluation of nested expressions in programming languages as Algol-68 or Ada should note that a different order of evaluation can be always achieved by the explicit use of control without descriptions. We have decided to base R-Maple strictly on predicate logic and thus we are *forced* to define descriptions in a unique way. Also note that descriptions of R-Maple are entirely within the realm of concrete syntax and have no independent semantic properties.

Descriptions allow a compact form of *Append* in a procedural form:

Append(*frs*,*scnd* | *res*) **is**
  if *frs* **eq** *nil* **then** *res*:=*scnd*
  else find [*hd*,*tl*]:=*frs* **in** *res*:=[*hd*,.*Append*(*tl*,*scnd*)]

Append in a clausal form is as follows.

Append(*nil*,*scnd* | *scnd*).
Append([*hd*,*tl*],*scnd* | *hd*,.*Append*(*tl*,*scnd*)).

The generator *Perm* permuting a list can be defined in a procedural way as follows.

Perm(*a* | *b*) **is**
  if *a* **eq** *nil* **then** *b*:=*nil* **else find** *Pickup*(*a* | *e*,*c*) **in** *b*:=[*e*,.*Perm*(*c*)]
Pickup(*hd*,*tl* | *a*) **is** *a*:=[*hd*,*tl*] **or find** *Pickup*(*tl* | *one*,*b*) **in** *a*:=[*one*,*hd*,*b*]

Both generators have this simple clausal forms:

Perm(*nil* | *nil*).
Perm(*a* | *e*,.*Perm*(*c*)) ← *Pickup*(*a* | *e*,*c*).
Pickup(*hd*,*tl* | *hd*,*tl*).
Pickup(*hd*,*tl* | *one*,*hd*,*b*) ← *Pickup*(*tl* | *one*,*b*).

We are now in position to give schemas of abbreviations for arithmetic operators occurring in expressions:

Mexpr+**Expr** ≡ .*Add*(**Mexpr**,**Expr**)
Mexpr−**Expr** ≡ .*Sub*(**Mexpr**,**Expr**)
Sexpr×**Mexpr** ≡ .*Mul*(**Sexpr**,**Mexpr**)
Sexpr div Mexpr ≡ .*Div*(**Sexpr**,**Mexpr**)
Sexpr rem Mexpr ≡ .*Rem*(**Sexpr**,**Mexpr**)

Generator *Fact* can be defined as

Fact(0 | 1).
Fact(*n* | *n*×.*Fact*(*n*−1)) ← *n* **ne** 0.

The reader is urged to compare this version with the one in Prolog.

14. Predefined Predicates.

R-Maple contains thirteen *predefined* predicates which can be invoked without explicit definitions. There are seven predefined tests

Eq,*Ne*,*Lt*,*Gt*,*Le*,*Ge*,*Print*

and six predefined generators with inputs:

. *Return*,*Add*,*Sub*,*Mul*,*Div*,*Rem*

Print is used to print out the values of results, the remaning tests are binary relations of comparisons. *Return* generates the value of its argument, the remaining generators are arithmetic generators with two inputs.

The meaning function correlates predefined predicates to the predicates of formal arithmetic with the same identifiers. We shall now give explicit definitions of the predicates in formal arithmetic. Then the intended meaning of predefined program predicates will be given by the meaning of the formal arithmetic predicates.

$\underline{Eq}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b}(\underline{x} = <\underline{a},\underline{b}> \ \& \ \underline{a} = \underline{b})$
$\underline{Ne}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b}(\underline{x} = <\underline{a},\underline{b}> \ \& \ \underline{a} \neq \underline{b})$
$\underline{Lt}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b}(\underline{x} = <\underline{a},\underline{b}> \ \& \ \underline{a} < \underline{b})$
$\underline{Gt}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b}(\underline{x} = <\underline{a},\underline{b}> \ \& \ \underline{b} < \underline{a})$
$\underline{Le}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b}(\underline{x} = <\underline{a},\underline{b}> \ \& \ \neg \ \underline{b} < \underline{a})$
$\underline{Ge}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b}(\underline{x} = <\underline{a},\underline{b}> \ \& \ \neg \ \underline{a} < \underline{b})$
$\underline{Print}(\underline{x}) \leftrightarrow \underline{x} = \underline{x}$
$\underline{Return}(\underline{x}) \leftrightarrow \underline{Eq}(\underline{x})$
$\underline{Add}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b} \exists \underline{c}(\underline{x} = <<\underline{a},\underline{b}>,\underline{c}> \ \& \ \underline{a} + \underline{b} = \underline{c})$
$\underline{Sub}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b} \exists \underline{c}(\underline{x} = <<\underline{a},\underline{b}>,\underline{c}> \ \& \ \underline{c} + \underline{b} = \underline{a})$
$\underline{Mul}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b} \exists \underline{c}(\underline{x} = <<\underline{a},\underline{b}>,\underline{c}> \ \& \ \underline{a} \times \underline{b} = \underline{c})$
$\underline{Div}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b} \exists \underline{c}(\underline{x} = <<\underline{a},\underline{b}>,\underline{c}> \ \& \ \neg \ \underline{a} < \underline{b} \times \underline{c} \ \& \ \underline{a} < \underline{b} \times (\underline{c}+1))$
$\underline{Rem}(\underline{x}) \leftrightarrow \exists \underline{a} \exists \underline{b} \exists \underline{c} \exists \underline{k}(\underline{x} = <<\underline{a},\underline{b}>,\underline{c}> \ \& \ Div(<<\underline{a},\underline{b}>,\underline{k}>) \ \& \ \underline{a} = \underline{b} \times \underline{k} + \underline{c})$

Note that the predicate *Print* is satisfied by any value of its argument. This suggests that if the program predicate *Print* serves any purpose at all , its usefulness must lie in the control [see (16)].

Although the form of the definitions is not the most suitable for direct computation, all predicates are primitive recursive. There is no need for the primitive recursive form of definitions because we do not intend to compute the predicates that way. We shall rather delegate the actual computation to the machine implementing R-Maple. However, the definitions of predicates are sufficient to *prove* the expected properties. And this is as it should be. The arithmetic counterparts of predefined predicates are stripped of any indication of the way how to compute them, they are intended merely for the meaning.

Note that the predicates can be satisfied only if supplied with the proper number of arguments: $\underline{Eq}(\underline{nil})$ and $Mul(<\underline{nil},6>)$ are false. The relations of integer division (*Div*) and of remainder (*Rem*) cannot be satisfied if the second argument is *0*. The first argument of subtraction (*Sub*) may not be less than the second one.

We shall adopt the same abbreviations for the predicates in the formal arithmetic as for invocations of program predicates. Thus $\underline{Add}(9,5 \mid \underline{x})$ stands for $\underline{Add}(<<9,5>,\underline{x}>)$.

As a straight-forward consequence of definitions we have:

$\underline{Lt}(\underline{a},\underline{b}) \leftrightarrow \underline{a} < \underline{b}$

similar equivalences are true of the other comparisons. This permits the expected correlation of comparison tests in R-Maple to the intended meaning:

$(\mathbf{Expr_1\ lt\ Expr_2})^* \leftrightarrow \mathbf{Expr_1^* < Expr_2^*}$

Similarly for other comparisons.

We also have

$(\mathbf{Var{:=}Expr})^* \leftrightarrow \mathbf{Var^* = Expr^*}$

15. Computation of Programs.

A computation of an R-Maple program **a** which does not contain processes (program counters) is performed in the environment of predicate definitions **d**. Definitions of predicates are used for the execution of invocations of defined predicates. A computation proceeds by steps which successively modify the original program **a**. The computation starts with the creation of a forward

process **a**!. A *computation sequence* is a possibly infinite sequence of programs

$$\mathbf{a}_0,\mathbf{a}_1,\mathbf{a}_2,\ldots,\mathbf{a}_n,\cdots \tag{1}$$

The first element in the sequence is **a**!, i.e. $\mathbf{a}_0 \equiv \mathbf{a}!$. Each successive element $\mathbf{a}_{i+1}$ is obtained from the preceding element $\mathbf{a}_i$ by an application of a *transformation*. A *transformation schema* is of the form

$$b \Rightarrow c$$

where b and c are terms possibly containing syntactic variables. A *transformation* is an instance of a transformation schema obtained by the substitution for all syntactic variables. A transformation $\mathbf{b} \Rightarrow \mathbf{c}$ can be applied to the program $\mathbf{a}_i$ if the program contains **b**. The term $\mathbf{a}_{i+1}$ is obtained by the replacement of the subterm **b** by the term **c** in $\mathbf{a}_i$.

A computation is called *sequential* if at most one transformation is applicable at any given time. If two or more transformations are applicable at the same time then the next element in a computation sequence can be obtained by the application of any of the applicable transformation. One of them is randomly chosen, thus achieving the effect of parallel computation and its associated non-determinism.

A computation sequence is called *terminating* if the sequence is finite. A computation sequence is *proper* if it terminates with the last term of the form **T**!. An infinite computation sequence (1) is *fair* if for every $i \geq 0$ there exist programs $\mathbf{b}_{i+1},\mathbf{b}_{i+2},\ldots,\mathbf{b}_n$ such that the computation sequence

$$\mathbf{a}_0,\mathbf{a}_1,\ldots,\mathbf{a}_i,\mathbf{b}_{i+1},\mathbf{b}_{i+2},\ldots,\mathbf{b}_n$$

is proper. The idea behind fair sequences is that although they do not terminate, they can be always stopped by fair scheduling of processes. Fair scheduling does not delay a process indefinitely. An improper computation sequence which terminates with the term $\mathbf{a}_n$ different from **T**! leads to a *deadlock*. Obviously, there is no transformation applicable to the last term $\mathbf{a}_n$. A computation of the program **a** in the environment of **d** is *proper* if each possible computation sequence is either proper or fair.

These definitions are meta-theoretic. The reader may have noticed certain analogy with the system of formal equations for recursive functions of Kleene [6]. By the proces of arithmetization, similar to the arithmetization of systems of formal equations, we shall introduce arithmetic predicates corresponding to the meta-theoretic ones. These predicates are introduced in Part II as they are needed for the definition of *semantics* of R-Maple.

Transformations are chosen in such a way that for each transformation $\mathbf{b} \Rightarrow \mathbf{c}$ other than invocation of a defined predicate we have as logical tautology $\mathbf{b}^* \leftrightarrow \mathbf{c}^*$. For the transformation of invocation of a defined predicate $\mathbf{b} \Rightarrow \mathbf{c}$ applied in the environment of definitions **d** we have as logical tautology $\mathbf{d}^* \rightarrow (\mathbf{b}^* \leftrightarrow \mathbf{c}^*)$. As a consequence of the Equivalence theorem we have for the computation sequence (1):

$$\mathbf{d}^* \rightarrow (\mathbf{a}_i^* \leftrightarrow \mathbf{a}_{i+1}^*)$$

as logical tautology. If the sequence (1) is proper with the final term $\mathbf{a}_n$ then, because of transitivity of equivalence, we have

$$\mathbf{d}^* \rightarrow (\mathbf{a}^* \leftrightarrow (\mathbf{T}!)^*)$$

In the view of

$$(\mathbf{a}^* \leftrightarrow (\mathbf{T}!)^*) \leftrightarrow (\mathbf{a}^* \leftrightarrow \forall_{\underline{z}}\, \underline{z}=\underline{z}) \leftrightarrow \mathbf{a}^*$$

we obtain the fundamental property of proper computations:

$$\mathbf{d}^* \rightarrow \mathbf{a}^* \tag{2}$$

Terms on LHS of transformations always contain program counters. Thus transformations are applicable only in the positions of a program where the "control" is currently present. The process "!" moves *forward* in a program; sequencing sequential operators and forking on parallel operators. The process **where** is used to *return* results obtained by invocations of predefined

generators in a "backward" movement inside the scope of a search (in the scope of the "existential" quantifier **find**). The process **move** allows the "return" of data still containing variables whose values are currently being searched for in "lazy" evaluation situations.

The computation, controlled by program counters, remains on the level of program operators. Expressions are passive in this respect. They are not "evaluated" even though they may contain variables. Program variables are removed from expressions by substitutions. It is assumed here that the operation of substitution for a program variable can be performed in one step.

Schemas of transformations required for execution of R-Maple programs are presented in the next two sections. The reader should not be disappointed by the relatively large number of transformations. Although there are many rules of transformations, they do not affect either the meaning of programs or the simplicity of the language. Whole groups of transformations have the same meaning; transformations within the same group differ only slightly in the control component.

It is quite easy to define a machine with only a few instructions, after all Turing Machines are of this kind. The large number of transformations in R-Maple is required for the efficiency of computation, just as real life computers tend to have many essentially similar, instructions in order to increase the speed of computations.

16. Transformations of Invocations.

When a forward process reaches an invocation: **Pred(Expr)**! then there are two possible courses of action depending on whether **Pred** is a predefined predicate or not. Let us deal with invocations of predefined predicates first.

Transformations for invocations of predefined predicates rely on the primitive-recursiveness of the named formal predicates. Transformations depend on a decision whether the named predicates of formal arithmetic are satisfied for certain arguments. If the predicate named by a predefined generator can be satisfied then there is a unique value of the output variable. For all predefined generators there is a primitive recursive function computing the output value from the input arguments. Thus we can safely delegate the decision of satisfiability and the computation of the generated value to the machine computing R-Maple in the knowledge that the machine can always proceed unassisted.

The computing machine executing identity tests *Eq*, *Ne* and split searches (see the next section) may be supplied with a quoted expression $\langle \underline{quoteop}, \mathbf{Lit} \rangle$ which must be split into two literals $\mathbf{Lit}_1$ and $\mathbf{Lit}_2$ such that

$$\langle \mathbf{Lit}_1, \mathbf{Lit}_2 \rangle = \mathbf{Lit}$$

The pairing operation and both of its inverse operations are primitive recursive. The *values* computed are unique but the *terms* denoting the values are not. For instance, if $\mathbf{Lit} \equiv \langle 3,5 \rangle$ then obviously $\mathbf{Lit}_1 \equiv 3$ but also $\mathbf{Lit}_1 \equiv \langle 1, \underline{nil} \rangle$.

If R-Maple programs rely on the complete typelessness of the language and mix numerals and pairs in a manner which would be considered a type error in other programming languages then the computing machine will have to do some computation of Gödel numbers as well. We are not saying that this is a good programming practice, but the invocation of $Add(4,\langle 1,nil \rangle \mid v)$ should come back with the value $v{:=}7$. Similarly the execution of the split search

find $[\, hd, tl \,]{:=}3$ **in Prog**

should find values for *hd* and *tl*, say $hd{:=}1$ and $tl{:=}nil$.

We leave it to the computing machine to come up with suitable literals when splits are required, and to convert between numeral and pair forms of literals. The form of literals can influence the efficiency of computations but never the meaning.

The four predefined ordering tests *Lt*,*Le*,*Gt*,*Ge* are computed as follows.

Ordering Transformation (**Pred** $\equiv$ *Lt*,*Le*,*Gt*,*Ge*):

$$\mathbf{Pred}(\langle \underline{quoteop}, \mathbf{Lit} \rangle)! \Rightarrow \begin{cases} \mathbf{T}! & \text{if } \mathbf{Pred}^{\circ}(\mathbf{Lit}) \\ \mathbf{F}! & \text{otherwise} \end{cases}$$

In order to justify this transformation we have to show that it leaves the meaning component invariant. Under the assumption **Pred***(**Llt**) we have

$$\mathbf{Pred}(<\underline{quoteop},\mathbf{Llt}>)!^{*} \equiv \mathbf{Pred}^{*}(\mathbf{Llt}) \leftrightarrow \forall_{x}\, x{=}x \equiv (\mathbf{T}!)^{*}$$

On the other hand, if $\neg$ **Pred***(**Llt**) then

$$\mathbf{Pred}(<\underline{quoteop},\mathbf{Llt}>)!^{*} \equiv \mathbf{Pred}^{*}(\mathbf{Llt}) \leftrightarrow \exists_{x}\, x{\neq}x \equiv (\mathbf{F}!)^{*}$$

The control will delay the process invoking an ordering predicate until the argument is a quoted expression. Then the computing machine will make the decision by computing the named primitive recursive formal predicate.

Note that the process invoking an ordering predicate with a consed expression, i.e. an expression still containing program variables, is delayed - because there is simply no applicable transformation - until a parallel process substitutes literals for all free variables. The last substitution will automatically transform the consed expression to a quoted one [see section (6)].

Predicates of identity *Eq* and *Ne* do not require both arguments to be fully reduced. This sometimes permits an earlier decision on still consed expressions:

Identity Transformations: $\mathbf{Expr}_1, \mathbf{Expr}_2$ *are not variables*

$$Eq(\mathbf{Expr}_1,\mathbf{Expr}_4)! \Rightarrow \begin{cases} Eq(\mathbf{Expr}_2,\mathbf{Expr}_5)! \;||\; Eq(\mathbf{Expr}_3,\mathbf{Expr}_6)! \\ \mathbf{T}! \\ \mathbf{F}! \end{cases}$$

$$\begin{cases} \text{if } \mathbf{Expr}_1^{*}{=}{<}\mathbf{Expr}_2^{*},\mathbf{Expr}_3^{*}{>} \;\&\; \mathbf{Expr}_4^{*}{=}{<}\mathbf{Expr}_5^{*},\mathbf{Expr}_6^{*}{>} \\ \mathbf{Expr}_1^{*} = \underline{nil} \;\&\; \mathbf{Expr}_4^{*} = \underline{nil} \\ \text{otherwise} \end{cases}$$

$$Ne(\mathbf{Expr}_1,\mathbf{Expr}_4)! \Rightarrow \begin{cases} Ne(\mathbf{Expr}_2,\mathbf{Expr}_5)! \;\mathbf{orp}\; Ne(\mathbf{Expr}_3,\mathbf{Expr}_6)! \\ \mathbf{F}! \\ \mathbf{T}! \end{cases}$$

$$\begin{cases} \text{if } \mathbf{Expr}_1^{*}{=}{<}\mathbf{Expr}_2^{*},\mathbf{Expr}_3^{*}{>} \;\&\; \mathbf{Expr}_4^{*}{=}{<}\mathbf{Expr}_5^{*},\mathbf{Expr}_6^{*}{>} \\ \mathbf{Expr}_1^{*} = \underline{nil} \;\&\; \mathbf{Expr}_4^{*} = \underline{nil} \\ \text{otherwise} \end{cases}$$

The justification of identity transformations is shown in a similar way as the justification of ordering transformations with the utilization of properties (4.1) and (4.2) of pairs.

When both sides of an identity test are pairs (consed or quoted) then control forks into two separate tests tests of the respective left-hand and right-hand sides. Note that there are no transformations for some equality tests as for instance x **eq** *nil* !. The execution cannot proceed and the process is delayed until a value of the program variable x is substituted by a transformation in a concurrent process.

Print Transformation:

$$Print(<\underline{quoteop},\mathbf{Llt}>)! \Rightarrow \mathbf{T}!$$

The transformation is trivially justified by the property of *Print*.

The sole purpose of the predefined test *Print* lies in its control component. The forward process invoking *Print* is delayed until its argument becomes a quoted expression. Then the literal is printed out and the invocation reduces to **T**!.

Predefined generator *Return* has the following transformation:

Return Transformation:

$$Return(\mathbf{Expr} \mid \mathbf{Var})! \Rightarrow \mathbf{T}! \;\mathbf{where}\; \mathbf{Var}{:=}\mathbf{Expr}$$

It is easy to see that the transformation preserves the invariance of meaning:

$$Return(\mathbf{Expr} \mid \mathbf{Var})^{*} \leftrightarrow \mathbf{Expr}^{*} = \mathbf{Var}^{*} \leftrightarrow$$

$$\forall_{\underline{x}}\, \underline{x}=\underline{x}\ \&\ \mathbf{Expr}^* = \mathbf{Var}^* \equiv (\mathbf{T!\ where\ Var{:=}Expr})^*$$

The significance of *Return* transformation lies in its control component. A **where** process which will proceed in parallel with the process **T!** is created. **T!** is delayed for the time being. The return process will try to bring the value of the variable **Var** back to the closest enclosing operator **find** on the same variable. Transformations governing the behaviour of **where** are given in the next section.

Arithmetic Transformations (**Pred** ≡ *Add,Sub,Mul,Div,Rem*):

$$\mathbf{Pred}(<\underline{quoteop},\mathbf{Lit}> \mid \mathbf{Var})! \Rightarrow \begin{cases} \mathbf{T!\ where\ Var{:=}}<\underline{quoteop},\mathbf{Lit}_1> & \text{if } \mathbf{Pred}^*(\mathbf{Lit},\mathbf{Lit}_1) \\ \mathbf{F!} & \text{if } \neg\, \exists \underline{a}\, \mathbf{Pred}^*(\mathbf{Lit},\underline{a}) \end{cases}$$

The justification of transformations is similar to the justifications for *Return* and ordering tests.

All five arithmetic generators require their input arguments quoted before the execution can proceed. When arithmetic predicates can be satisfied a return process **where** with the unique value satisfying the predicate is created.

Invocations of defined predicates are computed by the schema of *attribution* transformation. Let us assume that **Defs** stands for a list of definitions containing the definition of **Pred**:

Pred(Var) is Prog

Attribution transformations are the only ones which require predicate definitions:

Attribution Transformation (**Pred** *is defined in* **Defs**):

$$\mathbf{Pred(Expr)}! \Rightarrow \mathbf{Prog\{Var{:=}Expr\}}!$$

Let us show that the attribution preserves the meaning of programs relative to **Defs**. First of all we have as tautology

$$\mathbf{Defs}^* \rightarrow \forall \mathbf{Var}^*(\mathbf{Pred}^*(\mathbf{Var}^*) \leftrightarrow \mathbf{Prog}^*) \qquad (1)$$

As an instance of the substitution tautology of predicate logic we have

$$\forall \mathbf{Var}^*(\mathbf{Pred}^*(\mathbf{Var}^*) \leftrightarrow \mathbf{Prog}^*) \rightarrow (\mathbf{Pred}^*(\mathbf{Expr}^*) \leftrightarrow \mathbf{Prog}^*[\mathbf{Var}^*{:=}\mathbf{Expr}^*]) \qquad (2)$$

Because of the property (6.1) of substitution we have:

$$\mathbf{Prog\{Var{:=}Expr\}}!^* \leftrightarrow \mathbf{Prog}^*[\mathbf{Var}^*{:=}\mathbf{Expr}^*] \qquad (3)$$

As a consequence of (1), (2) and (3) we have:

$$\mathbf{Defs}^* \rightarrow (\ \mathbf{Pred}^*(\mathbf{Expr}^*) \leftrightarrow \mathbf{Prog\{Var{:=}Expr\}}!^*\ )$$

which justifies the transformation of attribution.

The control side of attributions is straight-forward. The process invoking an attribution finds the definition of the defined predicate in the list of definitions, replaces the invocation with a suitably instantiated body of the predicate, and continues its forward movement.

17. Unconditional Schemas of Transformations.

All transformations presented in this section preserve the intended meaning uncoditionally, i.e. they do not rely on lists of definitions.

We shall present the transformations in groups of related transformations. Whenever LHS and RHS of a transformation are mapped by "*" into different formulas we shall present the logical tautology assuring the invariance of computation under the meaning.

Negation Scheduling:

$$\mathbf{(not\ Prog)}! \Rightarrow \mathbf{not\ (Prog!)}$$

Conjunction Scheduling:

$$\mathbf{(Prog_1; Prog_2)}! \Rightarrow \mathbf{Prog_1!; Prog_2}$$

$$\mathbf{(Prog_1 \parallel Prog_2)}! \Rightarrow \mathbf{Prog_1!; Prog_2!}$$

Disjunction Scheduling:

$$\mathbf{(Prog_1\ or\ Prog_2)}! \Rightarrow \mathbf{Prog_1!\ or\ Prog_2}$$

(Prog$_1$ orp Prog$_2$)! => Prog$_1$! orp Prog$_2$!

Truth Table Transformations:

not (T!) => F!
not (F!) => T!
T!; Prog => Prog!
T! || Prog => Prog
F!; Prog => F!
F! || Prog => F!
Prog || T! => Prog
Prog || F! => F!
F! or Prog => Prog!
F! orp Prog => Prog
T! or Prog => T!
T! orp Prog => T!
Prog orp F! => Prog
Prog orp T! => T!
find Var in T! => T!
find Var in F! => F!

Both sides of scheduling transformations map into identical formulas. The justification for the truth table group lies in simple tautologies as $\forall_{\underline{x}}\, \underline{x}=\underline{x} \vee \mathbf{Prog}^* \leftrightarrow \forall_{\underline{x}}\, \underline{x}=\underline{x}$.

Note how two processes are created for parallel *ands* and *ors*. The execution of sequential *ands* and *ors* passes to the first argument and only if the truth value cannot be determined from the value of this argument the control moves on to the second one.

Decisions can be expressed with the help of sequential or parallel *ands*, *ors*, and *nots* but the efficiency of computations is improved by having *if op* as a basic operator.

Scheduling of Decisions:

(if Prog$_1$ then Prog$_2$ else Prog$_3$)! => if Prog$_1$! then Prog$_2$ else Prog$_3$

Transformations of Decisions:

if T! then Prog$_2$ else Prog$_3$ => Prog$_2$!
if F! then Prog$_2$ else Prog$_3$ => Prog$_3$!

Justification of the scheduling transformation is trivial, the other two rely on simple truth table tautologies.

Evaluation of a decision is sequential. The test is evaluated first; when it reduces to **T!** or **F!** then the parts **then** or **else** are evaluated respectively. If the test does not reduce to a truth value, the evaluation will either not terminate or terminates deadlocked.

Split Transformation: **Var$_1$,Var$_2$** *not free in* **Expr**; **Expr** *not variable*

(find [Var$_1$,Var$_2$]:=Expr in Prog)! =>

$$\begin{cases} \mathbf{Prog}\{\mathbf{Var}_1{:=}\mathbf{Expr}_1\}\{\mathbf{Var}_2{:=}\mathbf{Expr}_2\}! & \text{if } \mathbf{Expr}^* = \langle \mathbf{Expr}_1^*, \mathbf{Expr}_2^* \rangle \\ \mathbf{F}! & \text{otherwise} \end{cases}$$

The justification is based on the properties (4.1, 4.2) of pairs, (6.1) of substitution and on the tautology

$$\exists_{\underline{x}}(\underline{x} = \mathbf{s}\ \&\ \mathbf{A}) \leftrightarrow \mathbf{A}[\underline{x}{:=}\mathbf{s}] \quad \text{provided } \underline{x} \text{ is not free in } \mathbf{s} \qquad (1)$$

The execution of a split operation is delayed until its expression is not a program variable. A quoted expression can be split into constituents if its value is not equal to *nil*. A consed expression can be immediately split into two constituents. Constituents of the expression to be split are then substituted in **Prog** for the two variables **Var$_1$** and **Var$_2$**.

Search Scheduling:

(find Var in Prog)! => find Var in Prog!

The justification for the scheduling of searches is trivial.

An operation of search introduces a scope in which a value or values of the variable declared will be searched for. When the control reaches an invocation of a predefined generator then a return process, marked by the program counter **where** , is initiated. The return process will move the value of the generated variable backwards and upwards in the program until the value reaches its declaration.

The last groups of transformations control the backward movement of the value.

Backward Conjuction Group:

$Prog_1$ where Var:=Expr; $Prog_2$ => ($Prog_1$; $Prog_2$) where Var:=Expr
$Prog_1$ where Var:=Expr || $Prog_2$ => ($Prog_1$ || $Prog_2$) where Var:=Expr
$Prog_1$ || $Prog_2$ where Var:=Expr => ($Prog_1$ || $Prog_2$) where Var:=Expr

The transformations for the backward movement of a value through conjunctions rely on commutativity and associativity of conjunctions.

Backward Disjunction Group:

($Prog_1$ where Var:=Expr or $Prog_2$) or $Prog_3$ =>
$Prog_1$ where Var:=Expr or ($Prog_2$ or $Prog_3$)
($Prog_1$ where Var:=Expr or $Prog_2$) orp $Prog_3$ =>
$Prog_1$ where Var:=Expr orp ($Prog_2$! orp $Prog_3$)
($Prog_1$ where Var:=Expr orp $Prog_2$) or $Prog_3$ =>
$Prog_1$ where Var:=Expr orp ($Prog_2$ or $Prog_3$)
($Prog_1$ where Var:=Expr orp $Prog_2$) orp $Prog_3$ =>
$Prog_1$ where Var:=Expr orp ($Prog_2$ orp $Prog_3$)
($Prog_1$ orp $Prog_2$ where Var:=Expr) or $Prog_3$ =>
$Prog_2$ where Var:=Expr orp ($Prog_1$ or $Prog_3$)
($Prog_1$ orp $Prog_2$ where Var:=Expr) orp $Prog_3$ =>
$Prog_2$ where Var:=Expr orp ($Prog_1$ orp $Prog_3$)
$Prog_1$ orp ($Prog_2$ where Var:=Expr or $Prog_3$) =>
($Prog_1$ or $Prog_3$) orp $Prog_2$ where Var:=Expr
$Prog_1$ orp ($Prog_2$ where Var:=Expr orp $Prog_3$) =>
($Prog_1$ orp $Prog_3$) orp $Prog_2$ where Var:=Expr
$Prog_1$ orp ($Prog_2$ orp $Prog_3$ where Var:=Expr) =>
($Prog_1$ orp $Prog_2$) orp $Prog_3$ where Var:=Expr

The transformations for the backward movement of a value through disjunctions rely on commutativity and associativity of disjunctions.

Backtracking Group:

($Prog_1$ where Var:=Expr or $Prog_2$); $Prog_3$ =>
($Prog_1$; $Prog_3$) where Var:=Expr or $Prog_2$; $Prog_3$
($Prog_1$ where Var:=Expr or $Prog_2$) || $Prog_3$ =>
($Prog_1$ || $Prog_3$) where Var:=Expr orp $Prog_2$! || $Prog_3$
($Prog_1$ where Var:=Expr orp $Prog_2$); $Prog_3$ =>
($Prog_1$; $Prog_3$) where Var:=Expr orp $Prog_2$; $Prog_3$
($Prog_1$ where Var:=Expr orp $Prog_2$) || $Prog_3$ =>
($Prog_1$ || $Prog_3$) where Var:=Expr orp $Prog_2$ || $Prog_3$
($Prog_1$ orp $Prog_2$ where Var:=Expr); $Prog_3$ =>
$Prog_1$; $Prog_3$ orp ($Prog_2$; $Prog_3$) where Var:=Expr
($Prog_1$ orp $Prog_2$ where Var:=Expr) || $Prog_3$ =>
$Prog_1$ || $Prog_3$ orp ($Prog_2$ || $Prog_3$) where Var:=Expr
$Prog_1$ || ($Prog_2$ where Var:=Expr or $Prog_3$) =>
($Prog_1$ || $Prog_2$) where Var:=Expr orp $Prog_1$; $Prog_3$
$Prog_1$ || ($Prog_2$ where Var:=Expr orp $Prog_3$) =>
($Prog_1$ || $Prog_2$) where Var:=Expr orp $Prog_1$ || $Prog_3$
$Prog_1$ || ($Prog_2$ orp $Prog_3$ where Var:=Expr) =>
$Prog_1$ || $Prog_2$ orp ($Prog_1$ || $Prog_3$) where Var:=Expr

Backtracking transformations are based on the distributivity of conjunction over disjunction.

The control prepares for possible bactracking by routing the value being moved backward into one argument of a disjunction while the other argument is set to produce a value should the first argument fail.

Search Split Group:
find Var_1 in $Prog_1$ where Var_2:=Expr or $Prog_2$ =>
(find Var_1 in $Prog_1$ where Var_2:=Expr) or (find Var_1 in $Prog_2$)
find Var_1 in $Prog_1$ where Var_2:=Expr orp $Prog_2$ =>
(find Var_1 in $Prog_1$ where Var_2:=Expr) orp (find Var_1 in $Prog_2$)
find Var_1 in $Prog_1$ orp $Prog_2$ where Var_2:=Expr =>
(find Var_1 in $Prog_1$) orp (find Var_1 in $Prog_2$ where Var_2:=Expr)

Find split transformations are based on the tautology

$$\exists \underline{v}(A \vee B) \leftrightarrow \exists \underline{v}A \vee \exists \underline{v}B$$

When one result is backed up to a find operator with a possible alternative result then two searches are created. The one with the value being backed up can be immediately terminated because of the next transformation:

Successful Search: **Var** *is not free* **in Expr**
find Var in Prog where Var:=Expr => Prog{Var:=Expr}

Tautology (1) justifies this transformation. When a result is backed-up up to a **find** operator on the same variable then it is substituted for the variable in the scope of the operation. If the variable **Var** occurs free in **Expr** then the transformation is not applicable and the process **where** remains deadlocked.

Move Scheduling Group: $\neg\ Var_1 \equiv Var_2$
find Var_1 in Prog where Var_2:=Expr =>
move Var_1 beyond Var_2 in Prog where Var_2:=Expr

This transformation is justified trivially. If the result is on a different variable than the variable of a **find** then the operator **find** must be pushed back beyond the closest enclosing **find** on the variable of the result.

This happens when the expression **s** in **v:=s** contains variables, i.e. a partial result is being returned, and the backward movement reaches a **find** one of the variables possibly free in **s**. The variables in the term **s** cannot be moved out of its scope. Such enlargement of scopes happens in lazy evaluations.

The process **move** pushes **find** operators backwards until a **find** with the variable **move** looks for is reached. Last two groups of transformations control **move** processes.

Scope Enlargement Group:
(move Var_1 beyond Var_2 in $Prog_1$); $Prog_2$ =>
move Var beyond Var_2 in Prog; $Prog_2$
(move Var_1 beyond Var_2 in $Prog_1$) || $Prog_2$ =>
move Var beyond Var_2 in Prog || $Prog_2$
(move Var_1 beyond Var_2 in $Prog_1$) or $Prog_2$ =>
move Var beyond Var_2 in Prog or $Prog_2$
(move Var_1 beyond Var_2 in $Prog_1$) orp $Prog_2$ =>
move Var beyond Var_2 in Prog orp $Prog_2$
$Prog_2$ || move Var_1 beyond Var_2 in $Prog_1$ =>
move Var beyond Var_2 in $Prog_2$; Prog
$Prog_2$ orp move Var_1 beyond Var_2 in $Prog_1$ =>
move Var beyond Var_2 in $Prog_2$ orp Prog

If Var_1 is not free in $Prog_2$ then $Var \equiv Var_1$ and $Prog \equiv Prog_1$. On the other hand, if the variable Var_1 is free in $Prog_2$ then **Var** is the least variable not occuring in either of $Prog_1$,

Prog$_2$, and **Prog** $\equiv$ **Prog**$_1${**Var**$_1$:=**Var**}. The justification for the transformations of scope enlargement lies in the tautologies

$$\exists v A \,\&\, B \leftrightarrow \exists v(A \,\&\, B)$$
$$\exists v A \vee B \leftrightarrow \exists v(A \vee B)$$

where **B** does not contain the variable **v** free. The tautologies are always applicable after a possible renaming of bound variable **v** so it does not occur free in **B**.

Search Exchange Group: $\neg$ **Var**$_1 \equiv$ **Var**$_3$

find Var$_1$ **in move Var**$_2$ **beyond Var**$_1$ **in Prog** $\Rightarrow$ **find Var**$_2$**; Var**$_1$ **in Prog**

find Var$_1$ **in move Var**$_2$ **beyond Var**$_3$ **in Prog** $\Rightarrow$
move Var$_1$ **beyond Var**$_3$ **in move Var**$_2$ **beyond Var**$_3$ **in Prog**

First transformation relies on the tautology

$$\exists v \exists w A \leftrightarrow \exists w \exists v A$$

while the justification for the second transformation is trivial.

When a **move** process reaches an operator **find** declaring the variable of the return, the operators are swapped so a following process **where** can remove the swapped operator **find** . At the same time the **move** process is terminated by changing it back into a **find** . The second transformation creates a new **move** process by changing an outside operator **find** into a **move** so both **move** processes can proceed outwards until the first transformation is applicable.

18. Examples of Computation.

In the examples given below we shall use the symbol " $\Rightarrow$ " for *deterministic* computation in the transitive sense: $a \Rightarrow b$ means that there is a unique computation sequence of transformations initially applied to a and terminating in b. The symbol " $\approx\!\!>$ " is used for non-deterministic computations: $a \approx\!\!> b$ means that there are more computation sequences starting with a and at least one of them reaches b.

As the first example let us concatenate two lists and print-out the result:

Print(.*Append*([1,*nil*],[2,*nil*])

Obviously, the computation has to proceed in an environment of definitions including the definition of *Append*. The computation sequence given below is not complete. Some obvious steps are not shown. We have to stress that this sequence is only one of the possible computation sequences, because there is a parallelism involved in the return of values generated by *Append*:

Print(.*Append*([1,*nil*],[2,*nil*]) ! $\equiv$
(**find** *Append*([1,*nil*],[2,*nil*] | *a*) **in** *Print*(*a*))! $\equiv$
(**find** *a* **in** *Append*([1,*nil*],[2,*nil*] | *a*); *Print*(*a*))! $\Rightarrow$
find *a* **in** *Append*([1,*nil*],[2,*nil*] | *a*)!; *Print*(*a*) (1)

The only process in (1) is the forward process on the invocation of *Append*. In order to make this example more managable let us show just the computation of *Append* until it starts to affect the rest of the program. The reader has to bear in mind that the computation occurs inside of the whole program.

Append([1,*nil*],[2,*nil*] | *a*)! $\Rightarrow$
if [1,*nil*] **eq** *nil* ! **then** *a*:=[2,*nil*]
else find [*hd*,*tl*]:=[1,*nil*] **in** *a*:=[*hd*,.*Append*(*tl*,[2,*nil*])] $\Rightarrow$
(**find** [*hd*,*tl*]:=[1,*nil*] **in** *a*:=[*hd*,.*Append*(*tl*,[2,*nil*])]! $\Rightarrow$
x:=[1,.*Append*(*nil*,[2,*nil*])] ! $\Rightarrow$
find *b* **in** *Append*(*nil*,[2,*nil*] | *b*)!; *a*:=[1,*b*] $\Rightarrow$
find *b* **in** *b*:=[2,*nil*] !; *a*:=[1,*b*] $\Rightarrow$ (2)
find *b* **in T! where** *b*:=[2,*nil*]; *a*:=[1,*b*] (3)

The execution of the assignment *b*:=[2,*nil*] (2) creates a second process. Since there are no applicable transformations the forward process is delayed on **T!** in (3). The next transformation

is the backward conjunction of **where** :

find b **in T! where** $b:=[2,nil]$; $a:=[1,b]$ **=>**
find b **in** $(T!;\ a:=[1,b])$ **where** $b:=[2,nil]$

For the first time a non-deterministic choice has to be made. Two transformations are applicable: the reduction of conjunctions and the transformation of successful search.

find b **in** $(T!;\ a:=[1,b])$ **where** $b:=[2,nil]$ **≈>**
find b **in** $a:=[1,b]$ **! where** $b:=[2,nil]$ **≈>**
find b **in T! where** $a:=[1,b]$ **where** $b:=[2,nil]$ **=>**
T! where $a:=[1,2,nil]$

At this moment we have to show the computation in the context of the whole program because *Append* starts to affect it:

find a **in** $Append([1,nil],[2,nil]\ |\ a)!$; $Print(a)$ **≈>**
find a **in T! where** $a:=[1,2,nil]$; $Print(a)$ **≈>**
find a **in** $Print(a)!$ **where** $a:=[1,2,nil]$ **=>**
$Print([1,2,nil])!$ **=> T!**

The generator *Append* is a function and its computation proceeds without backtracking.

Split is a multi-valued generator. It will be used in the second example to demonstrate the backtracking. The following program tests whether the list $[1,2,nil]$ contains the list $[1,nil]$ as its initial sublist.

(find $Split([1,2,nil]\ |\ f,s)$ **in** f **eq** $[1,nil]$ **)! =>**
find a **in** $Split([1,2,nil]\ |\ a)!$; **(find** $[f,s]:=a$ **in** f **eq** $[1,nil]$**) =>**
find a **in** $(a:=[nil,[1,2,nil]]!$ **or Prog**$_1$**); Prog**

We have set

Prog$_1$ **≡ find** $[hd,tl]:=[1,2,nil]$; $Split(tl\ |\ fr,sc)$ **in** $a:=[[hd,fr],sc]$
Prog ≡ (find $[f,s]:=a$ **in** f **eq** $[1,nil]$**)**

The computation continues as follows.

find a **in** $(a:=[nil,1,2,nil]!$ **or Prog**$_1$**); Prog =>**
find a **in (T! where** $a:=[nil,1,2,nil]$ **or Prog**$_1$**); Prog =>** (4)
find a **in (T!; Prog) where** $a:=[nil,1,2,nil]$ **or Prog**$_1$**; Prog ≈>**
find a **in Prog! where** $a:=[nil,1,2,nil]$ **or Prog**$_1$**; Prog =>**
find a **in Prog! where** $a:=[nil,1,2,nil]$ **or (find Prog**$_1$**; Prog) =>** (5)
Prog!$\{a:=[nil,1,2,nil]\}$ **or (find** a **in Prog**$_1$**; Prog)**

The more interesting transformations are backtracking on (4), split of a search, and successful search on (5).

The overall backtracking situation should be obvious now: The computation is just about to try out whether the first pair of split lists satisfies **Prog**. If this were the case then there would be no need to execute the program

find a **in Prog**$_1$**; Prog**

which is held in the reserve. As it happens, the first alternative fails and the computation will fall back on the second alternative:

Prog!$\{a:=[nil,1,2,nil]\}$ **or (find** a **in Prog**$_1$**; Prog) =>**
nil **eq** $[1,nil]$ **! or (find** a **in Prog**$_1$**; Prog) =>**
F! or (find a **in Prog**$_1$**; Prog =>**)
find a **in Prog**$_1$**!; Prog =>**
find a **in (find** $Split([2,nil]\ |\ fr,sc)$ **in** $a:=[[1,fr],sc]$ **)!; Prog =>**
find a **in (find** b **in** $Split([2,nil]\ |\ b)!$; **(find** $[fr,sc]:=b$ **in** $a:=[[1,fr],sc]$**)); Prog ≈>**
find a **in** $(a:=[[1,nil],[2,nil]]$ **! or Prog**$_2$**); Prog ≈>**
$[1,nil]$ **eq** $[1,nil]$ **! or (find** a **in Prog**$_2$**; Prog) => T!**

where

Prog$_2$ ≡ (**find** b **in**
(**find** [hd,tl]:=[2,nil]; *Split*(tl | fr,sc) **in** b:=[[hd,fr],sc]);
(**find** [fr,sc]:=b **in** a:=[[1,fr],sc]))

When the last alternative of a generator with finite number of alernatives is exhausted, the generator fails. For example:

(**find** *Split*(nil | f,s) **in** f **eq** 3)! =>
find a **in** *Split*(nil | a)!; (**find** [f,s]:=a **in** f **eq** 3) =>
find a **in** (a:=[nil,nil] ! **or** (**find** [hd,tl]:=nil **in Prog**));
(**find** [f,s]:=a **in** f **eq** 3) ≈>
nil **eq** 3 ! **or** (find a **in** (**find** [hd,tl]:=nil **in Prog**); (**find** [f,s]:=a **in** f **eq** 3)) =>
find a **in** (**find** [hd,tl]:=nil **in Prog**)!; (**find** [f,s]:=a **in** f **eq** 3) =>
find a **in F**!; (**find** [f,s]:=a **in** f **eq** 3) => **find** a **in F**! => **F**!

where

Prog ≡ **find** *Split*(tl | fr,sc) **in** a:=[[hd,fr],sc]

Split is a *finite* generator. It can be backtracked into only finite number of times. An infinite generator can be backtracked into as many times as necessary. The simplest infinite generator is

Nums(i | n) **is** n:=i **or** *Nums*(i+ 1 | n)

Provided that *Nums* can be introduced with the intended meaning then one can prove by induction

Nums(i,n) ↔ $i \leq n$

Note that there would be no need to define *Nums* were it used just because of its meaning component; "i **le** n" would suffice. *Nums* is needed because of its control component. The invocation *Nums*(**Num** | n), when backtracked into sufficiently many times, generates into n the natural numbers

Num*,**Num***+ 1,**Num***+ 2, · · ·

Nums is used in another infinite generator *Primes* which generates all primes:

Primes(| n) **is find** *Nums*(2 | i) **in** *Isprime*(i); n:=i

with the primality test *Isprime* introduced as follows:

Candiv(i,n | o) **is** $i \times i$ **lt** n; (o:=i **or** *Candiv*(i+ 1,n | o))
Isprime(n) **is not** (**find** *Candiv*(2,n | i) **in** n **rem** i **eq** 0)

Primes is a filter generating only those values of *Nums* which are primes. Although *Primes* and *Isprime* differ vastly in control, their intended meanings are easily proven equivalent:

Primes(n) ↔ *Isprime*(n)

The intended meanings of *Candiv* and *Isprime* are:

Candiv(i,n | o) ↔ $o \times o < n$ & $i \leq o$ (6)
Isprime(n) ↔ ∀i(*Canprime*(2,n,i) → ¬ *Rem*(n,i,0)) (7)

The equivalence (6) is obtained by induction from the intended meaning of *Candiv*.

The predicate *Isprime* is a good example of the importance of control. From the point of view of pure meaning the primality test could have been defined in arithmetic as

Isprime(n) ↔ ∀i (*Rem*(n,i,0) → i = 0 ∨ i = 1 ∨ i = n) (8)

The auxiliary generator *Candiv* is required to turn the unbounded universal quantifier to a bounded one so *Isprime* can be computed. Efficiency is improved by testing the candidate divisors of n only up to the root of n. The proof that (7) and (8) are equivalent is straight-forward.

When executing programs with infinite generators one has to have at least a certain degree of confidence that the termination condition will be eventually satisfied.

It is an open problem of number theory whether there is an infinite number of prime twins as for instance *17* and *19*. One would probably have some difficulties with the proof that the test for the second twin $Isprime(n+2)$ in the following program will become eventually satisfied for a large prime.

find $Primes(\mid n)$ **in** n **gt** 100000000; $Isprime(n+2)$; $Print(n)$

The bounded search in the above program is inherently sequential: first a prime will be found then it will be tested for the terminating conditions. Primality tests for both candidate twins can be speeded up by a parallel computation:

(**find** n **in** $Primes(\mid n)$ || n **gt** 100000000; $Isprime(n)$; $Print(n)$)! $\approx\!>$
find n **in** $Primes(\mid n)$! || **Prog** $\approx\!>$
find n **in** (**T**! **where** n:=**Num or Prog**$_1$) || **Prog** $\approx\!>$
find n **in** (T! || **Prog**) **where** n:=**Num orp Prog**$_1$! || **Prog** $\approx\!>$
Prog{n:=**Num**} **orp** (**find** n **in Prog**$_1$! || **Prog**)

where **Num** stands for a prime and

Prog $\equiv$ n **gt** 100000000!; $Isprime(n+2)$; $Print(n)$
Prog$_1$ $\equiv$ (**find** $Nums$(**Num**+ 1 | i) **in** $Isprime(i)$; n:=i)

The search **find** n **in Prog**$_1$; **Prog** would be in sequential execution held back in reserve to fall back into when **Prog** fails. Parallel "backtracking", if the term is appropriate at all, invokes the search immediately.

Note that nothing can be gained by having the terminating conditions in **Prog** connected by parallel *ands*. Neither of the tests can advance very much unless n has been replaced by a quoted literal.

The examples of both finite and infinite generators hopefully demonstrated the elegance of our concept of backtracking. The backtracking in R-Maple is made explicit and straight-forward. Furthermore, one can immediately see that no alternative is lost. This is due to the invariance of the meaning component. Contrast this with the obscurity of backtracking hidden in interpreters of programming languages as Prolog.

Let us present three examples where Prolog suffers because of its intentional negligence of control:

$P(x) \leftarrow Q(x),!,R(x)$
$P(x) \leftarrow S(x)$
$\leftarrow P(a)$

This is a very common case where programmers rely on implicit control built into Prolog interpreters. Let us assume that during the refutation of $P(a)$ the test $Q(a)$ is satisfied but $R(a)$ fails. The next clause for $P(x)$ should be now tried. But the programmer does not want to try the second clause once $Q(a)$ has been satisfied. He has really meant the second clause to be

$P(x) \leftarrow \neg\, Q(x),S(x)$

but because of the difficulties with negation in Prolog clauses, and for obvious reasons of efficiency, he places a *cut* in the first clause barring the backtracking into the second clause once past $Q(x)$. Cuts cannot be explained in logic within the context of a clause and without an ordering of clauses.

R-Maple acknowledges this very common situation and permits to write

$P(x)$ **is if** $Q(x)$ **then** $R(x)$ **else** $S(x)$

This has the meaning as if the second Prolog clause for $P(x)$ has started with $\neg\, Q(x)$ but without the inefficiency of double evaluation of $Q(x)$.

The second example:

$\leftarrow G(a,y),!,P(y)$

The programmer knows that the relation $G(a,y)$ is a function producing a unique y. He does not want to backtrack into G should $P(y)$ fail.

We simply write $P(.G(a))$ or if we want to be explicit

find $G(a \mid y)$ **in** $P(y)$

and know that no backtracking into G can occur if G is a functional generator where no assignment to y is moved back over a disjunction.

The third example demonstrates how the execution of Horn clauses suffers because the quantifiers have been moved into prenex forms by Skolemization.

$P(a)$
$P(x) \leftarrow A(x)$
$Q(x) \leftarrow R(x,y)$
$\leftarrow P(x),Q(x),S(x)$

Standard Prolog interpreters unify the last clause with the first one to produce

$\leftarrow Q(a),S(a)$

Now all natural numbers y in the third clause will be tried until, say, $R(a,100)$ succeeds; but alas $S(a)$ fails. The interpreter knowing nothing about the scopes of searches will backtrack into $R(a,101)$ and possibly go into an infinite search, although only the backtrack to the second clause for $P(x)$ can find another x satisfying $S(x)$.

R-Maple programs retain the full indication of scopes and the computation will go as expected:

$P(\mid x)$ **is** $x{:=}a$ **or** $A(\mid x)$
$Q(x)$ **is find** $Nums(0 \mid y)$ **in** $R(x,y)$

(**find** $P(\mid x)$ **in** $Q(x)$; $S(x)$)! $\approx\!>$ $Q(a)!$; $S(a)$ **or Prog** $\approx\!>$
($R(a,100)!$ **or** (**find** $Nums(100+1 \mid y)$ **in** $R(x,y)$)); $S(a)$ **or Prog** $\approx\!>$
T!; $S(a)$ **or Prog** $\approx\!>$ $S(a)!$ **or Prog** $\approx\!>$ **F**! **or Prog** $\approx\!>$ **Prog**!

where

Prog $\equiv$ (**find** x **in** $A(\mid x)$; $Q(x)$; $S(x)$)

The reader is urged to program the predicate *Isprime* as another example of difficulties with control in Prolog.

Prolog has been designed as a language with programs concerned mainly with the meaning component. No matter how sophisticated the theorem prover in the interpreter of Prolog is, the overal efficiency can be assured only by explicit control.

This can be seen from the above examples as well as from any large Prolog program which must be heavily infested with cuts in order to assure tolerable efficiency.

We have built the control component directly into our programming language. As a consequence, a programmer can explicitly visualize and direct the ordering of execution. When this control component is suppressed by a programming language then, almost invariably, two things will happen.

1) Programmers will quickly master the scheduling strategy of the local interpreter and adjust their supposedly purely logical programs accordingly. When such a program is transferred to an environment with a "dumb" interpreter, the program will perform less efficiently if at all.

2) The implementors of interpreters with automatic control component will invariably make some control mechanisms explicitly available. Now, what is worse: A programming language with explicit standard control or a language proudly claiming that the meaning is of overriding importance only to be brought down to the earth by programs heavily infested with non-standard control?

The first point seems to be also relevant to the growing tendency to suppress the control in modern programming languages. The freedom so generously profferred to the implementors of compilers for such languages can be the undoing of many a good program. Here again, our

approach seem to be a sound one: we provide for high level constructs such as descriptions and bounded searches which do not require that the programmers explicitly specify the control, but at the same time we make the control well defined. Actually we are forced to do so by the requirements of logic.

Some functional programming language permit infinite data structures, such as lists and trees, to be used in connection with lazy evaluation. For instance, one can define an infinite list of all primes and use it in subsequent computations only partially.

There is nothing special about logical predicates satisfied by infinitely many individuals. It is certainly less appealing to introduce infinite individuals into the universe of discourse to play the role of infinite predicates and then to use the infinite objects only finitely.

The entire concept of computation rests on the concept of *finiteness*. To start with a very powerful theory with models requiring infinite elements which are used finitely seems to us a slight overkill. Mathematical induction rests on the downward finiteness of the sequence of natural numbers. What complicated forms of induction are required to prove properties of infinite objects which are never used in their entirety?

As the last example we shall demonstrate the use of expressions still containing variables. The generator *Insert* inserts a natural number into an ordered list:

```
Insert(n,lst | nlst) is
  if lst eq nil then nlst:=[n,nil]
   else find [hd,tl] in lst
    if n lt hd then nlst:=[n,lst]
     else if n eq hd then nlst:=lst
     else find aux in nlst:=[hd,aux]; Insert(n,tl | aux)
```

Note that the assignment to the result in the last line is executed before the recursive invocation of *Insert*. One would normally expect the last line to be

```
     else nlst:=[hd,.Insert(n,tl)]
```

Insert, as given above, permits an insertion of two or more elements in parallel:

```
( find b in ( find a in Insert(5,[3,lst] | a) || Insert(4,a | b) ); Print(b) )! =>>
find b in ( find a in
  ( find aux in a:=[3,aux] !; Insert(5,lst | aux) ) || Insert(4,a | b)! ); Print(b) =>>          (9)
find b in ( find a in
  ( find aux in Insert(5,lst | aux)! where a:=[3,aux] ) || Insert(4,a | b)! ); Print(b) =>>
find b in ( find a in
  ( move aux beyond a in Insert(5,lst | aux)! where a:=[3,aux] ) ||
    Insert(4,a | b)! ); Print(b) =>>
find b in ( find a in
  move aux beyond a in ( Insert(5,lst | aux)! || Insert(4,a | b)! ) where a:=[3,aux] );
    Print(b) =>>
find b in
  ( find aux; a in ( Insert(5,lst | aux)! || Insert(4,a | b)! ) where a:=[3,aux] );
    Print(b) =>>
find b in ( find aux in Insert(5,lst | aux)! || Insert(4,[3,aux] | b)! ); Print(b)            (10)
```

The first invocation of *Insert* has constructed the partial result [3,*aux*] in (9) which is passed to the second invocation of *Insert* in (10). The second *Insert* will be in practice delayed on the first decision inside its body while it waits for the partial result. After the partial result is substitued, all decisions in the second invocations can be performed before the process might be delayed again in the recursive invocation to itself.

19. Conclusions of Part I.

We have decided not to include integers to R-Maple. Natural numbers as the starting point, make the intended meanings of predicate definitions simpler. R-Maple programs should be defined over the whole universe. This is because of the universal quantifier in the intended meaning of predicate definitions. The domain of pairs plus *nil* coincides with the domain of natural numbers. Thus a program operating on lists is also defined for all natural numbers and vice versa. Had we started with integers, then the negative numbers would not correspond to pairs and the universal quantifier would not express the intended meaning of programs as, for instance, *Append* which is designed to operate only on lists.

This is not a serious difficulty, however. In the Part II we shall show how to restrict the universal quantifiers to apply only to elements of certain types. With R-Maple extended by types the underlying formal theory can be the theory Z of integers, or even better the theory R of rational numbers.

Other basic types as characters and strings can be formally introduced by embedding into natural numbers.

For reasons of keeping this report simple, we have adapted the position that, during the computation of invocations and split searches, the R-Maple machine performs the substitution in one step. This position is slightly unrealistic with the present day hardware. Such an understanding of substitution involves a significant amount of copying and also costs time. In practice the substitution will not be done but rather an *environment* will be maintained during the computation. The environment will carry the bindings of variables to the terms to which they have to be substituted. Alternatively, the substitution can be performed via *combinators* [5,16].

Both schemes are acceptable as long as that they support the effect of a *virtual* one-step substitution.

For the reasons of simplicity we have decided against the inclusion of higher-order predicates. Higher order predicates accept other predicates as arguments. A predicate argument can be then *attributed* inside the body of a higher-order predicate. The definitions of predicates are terms which can be treated as data. Thus there is almost no problem, at least from the computational point of view, with the introduction of higher-order predicates. However, the list of predicate definitions would dynamically change during the execution of programs.

Predicates can yield another predicates as the values of output variables even now. This is possible because definitions of predicates are just literals. Some syntax sugaring is required before the generated predicates can be written in an elegant way.

Predicates of higher-order would necessitate more profound changes in the semantics. These changes will be outlined in the Part II. For the time being we lack the required formal apparatus.

Another "feature", not treated in this report, is some sort of data base providing for the environment in which one can store, retrieve, and execute both the general data, as well as the predicates which are data of a special sort.

Such a data-base is currently the subject of research although the present author has some ideas of how to structure the environment. The ideas come from his language Maple [17] which is the predecessor to R-Maple.

A final word on the expected performance of an R-Maple machine: Although R-Maple is more powerful than Prolog because of its explicit control, and it is also stronger than LISP because of its non-determinism, it does not require a sophisticated interpreter. The reason for this is that the backtracking is expressed in the computation rules rather than built into the interpreter. Furthermore, there is no need for a unification scheme which is so costly in Prolog. We do not see a reason why a good implementation should not be comparable to the implementations of LISP.

[1] Clark K. L., McCabe F. G., Gregory S., IC-Prolog Reference Manual; Research Report Imperial College, London 1981.

[2] Friedman D. P., Wise D. S., CONS should Not Evaluate its Arguments; in Automata, Languages and Programming (Michelson, Milner eds.), Edinburgh University Press 1976.

[3] Henderson P., Functional Programming Application and Implementation; Prentice-Hall, Englewood Cliffs 1980.

[4] Hilbert D., Bernays P., Grundlagen der Mathematik I, II; Springer-Verlag, Berlin 1968.

[5] Hindley J.R., Lercher B., Seldin J. P., Introduction into Combinatory Logic, Cambridge University Press, 1972.

[6] Kleene S., Introduction to Metamathematics; North-Holland, Amsterdam 1971.

[7] Kowalski R., Logic for Problem Solving; North Holland, Amsterdam 1979.

[8] Mauer W. D., A Programmer's Introduction to Lisp; American Elsevier, Amsterdam 1973.

[9] Naur P. (Ed.), Revised Report on the Algorithmic Language Algol 60; Regnencentralen, Copenhagen, 1962.

[10] The Programming Language Ada, Reference Manual; Lect. Notes in Comp. Science, Springer-Verlag 1981.

[11] Rivest R., Shamir A., Adleman L., A Method for Obtaining Digital Signatures and Public Key Cryptosystems; CACM February 1978.

[12] Scott D., Data Types as Lattices. Siam J. Comp. 1976.

[13] Scott D., Logic and Programming Languages, CACM 1977.

[14] Shoenfield J. R., Mathematical Logic, Addison-Wesley, Reading 1967.

[15] Stoy J., The Scott-Strachey Approach to the Mathematical Semantics of Programming Languages, Project MAC, MIT, 1974.

[16] Turner D. A., A New Implementation Technique for Applicative Languages, Software Practice and Experience 1979.

[17] Voda P. J., Maple: A Programming Language and Operating System; Proc. of ACM Symp. on POPL, Albuquerque, 1982.

[18] van Wijngaarden A. (Ed.), Revised Report on the Algorithmic Language Algol 68; Sigplan Notices, May 1977.

[19] Wirth N., The Programming Language Pascal; Acta Informatica 1971.