Stanford Verification Group
Report No. 11

Computer Science Department
Report No. STAN-CS-79-731

March 1979
Edition 1

LEVEL II

STANFORD PASCAL VERIFIER USER MANUAL

by

STANFORD VERIFICATION GROUP

AD A071900

Research sponsored by

Advanced Research Projects Agency

DDC
RECEIVED
JUL 26 1979
C

COMPUTER SCIENCE DEPARTMENT
Stanford University

DDC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.



79 07 16 006

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-79-731, PVG-11	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Stanford Pascal Verifier User Manual		5. TYPE OF REPORT & PERIOD COVERED technical
		6. PERFORMING ORG. REPORT NUMBER PVG-11
7. AUTHOR(s) Stanford Verification Group		8. CONTRACT OR GRANT NUMBER(s) MDA903-76-C-0206
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory Stanford University Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2494
11. CONTROLLING OFFICE NAME AND ADDRESS Eugene Stubbs ARPA/PM 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE March 1979 13. NUMBER OF PAGES 117
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Philip Surra, ONR Representative Durand Aeronautics Building, Rm 165 Stanford University Stanford, CA 94305		15. SECURITY CLASS. (of this report) 15 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without limitation on dissemination

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Stanford Pascal verifier is an interactive program verification system. It automates much of the work necessary to analyze a program for consistency with its documentation, and to give a rigorous mathematical proof of such consistency or to pin-point areas of inconsistency. It has been shown to have applications as an aid to programming, and to have potential for development as a new and useful tool in the production of reliable software.

This verifier is a prototype system. The purpose of this manual is to introduce the verifier to a wider group of users for experimentation.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SVG

Stanford Verification Group
Report No. 11

March 1979
Edition 1

Computer Science Department
Report No. STAN-CS-79-731, SVG-11

(14)

(6)

STANFORD PASCAL VERIFIER USER MANUAL.

by

(10)

STANFORD VERIFICATION GROUP

D.C. LUCKHAM, S.M. GERMAN, F.W. HENKE, R.A. KARP, P.W. MILNE, D.C. OPPEN, W. POLAK, W.L. SCHERLIS

(12) 127 p'

(9) Technical repts,

(11) Mar 79

(15) MDA903-76-C-0206, ARPA Order-2494

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

P.W. Milne is employed by CSIRO Division of Computing Research, P.O. Box 1800, Canberra City ACT 2601, Australia

094 120

CONTENTS

PART I

Foreword 1
1. Overview 3
2. The Verifier 3

2.1 VCG 4
2.2 The theorem prover 5

3. The Assertion Language 6

3.1 Kinds of assertion statements 6
3.2 Data structure terms 7

4. The Rule Language 10

4.1 Backward rules 10
4.2 Replacement rules 10
4.3 Forward rules 10
4.4 Differences between rules 11
4.5 Rules for data structure terms 11

5. Verification Examples 12

5.1 First example: understanding VCs 13
5.2 Concepts, documentation and verification 16
5.3 A hard invariant 17
5.4 Defining concepts to document a program 21
5.5 Specifications for sorting 23
5.6 A pointer example 24
5.7 Verification of Pascal list structure operations 25
5.8 A larger example 27

PART II

Chapter 1 Differences from Standard Pascal 35

1.1 Comments 35
1.2 Program files 35
1.3 Procedure definitions 35
1.4 Assertions 36

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

Contents

1.5	Blocks	36
1.6	Types	37
1.7	Functions	37
1.8	Input/Output	37
1.9	Global variables	38
1.10	Virtual variables and Passive statements	38
1.11	Operator precedence	38
1.12	Union types	39
Chapter 2	User Commands	42
2.1	Imperative commands	42
2.2	Setting system parameters	45
2.3	Query commands	46
2.4	System control	46
Chapter 3	Description of the Simplifier	48
3.1	Introduction	48
3.2	Prover for arithmetic	48
3.3	Record prover	49
3.4	Array prover	50
3.5	List structure prover	50
3.6	Remarks	50
Chapter 4	The Rule Language	51
4.1	Introduction to rules	51
4.2	Using the rule language	60
References		75
Appendix A	Syntax Charts	A-1
Appendix B	Parser Error Messages	B-1
Appendix C	VCG Axiomatic Semantics	C-1

FOREWORD

The Stanford Pascal verifier is an interactive program verification system. It automates much of the work necessary to analyze a program for consistency with its documentation, and to give a rigorous mathematical proof of such consistency or to pin-point areas of inconsistency. It has been shown to have applications as an aid to programming, and to have potential for development as a new and useful tool in the production of reliable software.

This verifier is a prototype system. It has inadequacies and shortcomings. It is undergoing continuous improvement, and is expected to be used eventually in conjunction with other kinds of program analyzers. The purpose of this manual is to introduce the verifier to a wider group of users for experimentation. We hope to encourage both feedback to help improve this system, and the development of other program analyzers.

The verifier is coded in Maclisp, a version of Lisp developed at M.I.T. for PDP-10 computers. Versions of the verifier run under the TOPS-20 operating system and the Stanford WAITS operating system.

How to read this manual

The manual is divided into two parts. Notation based on the SAIL character set is used throughout because it is closer to mathematical usage. The alternate notation based on ASCII is sometimes indicated; the reader can always find the corresponding ASCII notation by refering to Appendix A.

Part I is an introduction to the verifier. It contains a short survey of its features and components, and examples of its use. The reader who has completed Part I should be able to construct simple examples and run them. He should also have gained some idea of what the verifier can do and what inadequacies to expect.

Part II is a manual for those users who embark upon serious experiments with the verifier. Chapter 1 lists the differences between standard Pascal and the documented Pascal that the verifier requires as input. The major differences are the required documentation. There are also some minor differences in code. This is because it is planned that the verifier will accept a more general programming language, Pascal Plus, including Modules and constructs for concurrent processing. There is no discussion of the extended language in this manual.

Chapter 2 describes the toplevel user commands.

Chapter 3 is a short description of the special purpose theorem provers. This tells the user what kinds of knowledge are "built in" and what he must describe to the verifier by means of rules.

Chapter 4 is about the Rule Language. This chapter is in two sections. The first describes the rule language and how to express mathematical facts as rules; the second section gets into the intricacies of writing rules and why rules written one way may lead the verifier into much more efficient proof searches than if they are written another way. Section 1 should be enough for many simple examples.

Appendix A contains syntax charts similar to the charts given in the Pascal User Manual. Here one will find the syntax of user commands for running the verifier and the syntax of input to the verifier, i.e., programs, assertions, and rules. Also, at the beginning of Appendix A, the alternate ASCII notation for mathematical symbols is given. Appendix B is a list of parser error messages with a more detailed description of their meaning than is provided by the comments from the system. Appendix C presents the axiomatic semantics used by the verification condition generator.

Acknowledgements

We would like to acknowledge the contributions made to the development of the verifier by our colleagues Shigeru Igarashi, Ralph London, Nori Suzuki, Scot Drysdale and Greg Nelson.

PART I

INTRODUCTION TO THE STANFORD PASCAL VERIFIER

1. Overview

Section 2 gives a toplevel overview of the verifier and how it is used. Section 3 describes the assertion language, the language in which the specifications of a program and the accompanying internal inductive assertions must be written. There are some brief remarks about what kinds of internal inductive assertions are required. A full description of compulsory assertions is given in Part II, Chapter 1, and this information is also contained in the syntax charts. Section 4 outlines some of the basic constructs of the rule language. Rules defining concepts used in assertions must be written in the rule language. Part II, Chapter 4 gives more details about rules and how the theorem prover uses them. Section 5 gives a number of examples illustrating the use of the verifier. The first few are quite simple and should be sufficient to enable the reader to run some simple examples of his own. The final example, on verifying a parser, illustrates formulation of rules from mathematical theories and the use of the verifier in debugging and improving specifications. At this point the reader is in a position to begin finding his own ways to use the verifier. The methodology of using verification systems is by no means fully explored. Further examples of verification experiments are given in the references at the end of Part II.

2. The Verifier

The verifier employs the inductive assertion method due to Floyd [7] for reasoning about programs. Floyd's method was developed into a logic of programs by Hoare [11] and others [3, 14]. The verifier constructs its proofs within this logic of programs. It requires as input a Pascal program together with documentation in the form of inductive assertions at crucial points in the program and ENTRY/EXIT assertions attached to each procedure.

Fig. 1 shows what happens when the programmer gives this input to the verifier. The input goes first to a verification condition generator which gives as output a set of purely logical conditions called Verification Conditions (VCs). There is a VC for each path in the program. If all of the VCs can be proved, the program satisfies its specification. The next step is to try to prove the VCs using various algebraic simplification and proof methods. Those VCs that are not proved are displayed for analysis by the programmer. If a VC is incorrect, this may reveal a bug in the program or insufficient documentation at some point. A modification is made to the input and the problem is rerun. If the unproven VCs are all correct this merely indicates that the proof procedures need more mathematical facts about the problem. The programmer then specifies appropriate lemmas as as rules using the Rule Language. These rules are input to the verifier and the proof is attempted again. Ideally, the time for a complete cycle (Fig. 1) in a modern interactive computing environment should be on the order of a minute for a one page program.

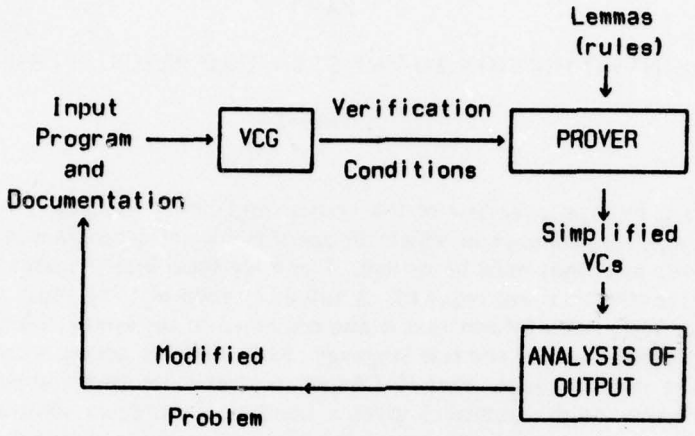


Fig. 1.

2.1 VCG

VCG contains a parser and a Verification Condition Generator (VCGEN). VCGEN uses axiomatic semantics of the programming language to generate VCs. We chose Pascal because at the time this project began it was the only language for which such an axiomatic semantics within Hoare's logic of programs had been given [13]. VCGEN simply takes the place of the code generator in a compiler. The program together with inductive assertions is parsed for syntax and type compatibility (see Part II, Chapter 1 for details). The result is an internal tree representation from which the VCs are constructed by transforming the inductive assertions as a function of the code. The transformations correspond to axiomatic proof rules defining the meaning of the programming language constructs. The theory of VCGEN is presented in [14].

The important point is that if all of the VCs can be proved then there is a proof within the weak logic of programs that the given program satisfies its ENTRY/EXIT assertions and also that each subsection of the program satisfies its surrounding inductive assertions. Such a proof can be constructed by reversing the transformations that were applied by VCGEN. So, the VCs are sufficient conditions for correctness, but not always necessary ones.

The truth of the VCs often depends on how completely the inductive assertions describe sections of the code. As a matter of practical convenience, the programmer should not be forced to supply documentation beyond what is necessary to understand the program. The transformations currently used by VCG are combinations of the axiomatic semantic rules of Pascal. The objective of such combinations is to reduce the number of situations in which the user has to repeat his assertions in trivial and tedious ways (this was a problem with earlier verifiers). The basic

assertion requirements are that procedure and function declarations must have ENTRY/EXIT specifications, and loops within the body of the program must have invariant assertions. It is not necessary to place assertions at all GOTO labels. There are other required assertions (e.g., for global variables) and the details of these are in Part II, Chapter 1.

It is easy to modify VCG for other languages that have axiomatic semantics formalizable within the logic of programs. No other component of the verifier depends on the input programming language.

2.2 The theorem prover

The prover takes a verification condition and attempts to prove it correct. If it succeeds, it returns *TRUE*; if it proves that the verification condition is inconsistent, it returns *FALSE*; if neither, it returns a simplified version of the verification condition.

The prover is the most complex component of the verifier. The major issue in its design is the trade-off between generality (i.e., logical completeness) and its average response time to given problems. If the theorem prover is very general, it takes too long to prove VCs and the user gives up waiting. If it is too restricted in its logical power and requires to be told too many trivial facts (e.g., $x + 1 \leq y \rightarrow x < y$) the user will quickly become frustrated.

We have tried to solve this problem by separating the prover into two parts. The first part, called the "simplifier", contains built-in knowledge about the most common data structures of programming languages -- numbers, arrays, records, list structure, and simplifies very quickly expressions involving these data structures. The second part of the prover is the "rulehandler", which uses user-supplied axioms to reason about data structures not handled by the simplifier. The simplifier is thus a very efficient but very specialized prover while the rulehandler is very general and not necessarily very efficient. How the two components coexist is a mystery to the authors of this manual.

As we shall see in Part II, Chapter 3, the simplifier includes a decision procedure for the quantifier-free theory of rationals, arrays, records, list structure and uninterpreted function and predicate symbols under +, ≤, store and select, cons, car and cdr. The main pitfall with a built-in simplifier such as this is that it is in fact "built in" -- its workings are hidden from the user.

The rulehandler accepts rules supplied by the user to define the concepts used in documenting his program. These rules are treated as defining axioms for these concepts and are automatically used by the prover in searching for a proof. The language for stating rules allows the user to supply hints on how the rule is to be used. This is one method of making the search for a proof more efficient (see Bledsoe [2]). It is possible to write a set of mathematical facts as a set of rules in different ways, some resulting in much more efficient behavior from the rulehandler than others. Also, sufficient mathematical facts for a proof may be supplied, but, depending on how they are expressed as rules, the rulehandler may or may not succeed in finding a proof. In Section 4 we briefly summarize the kinds of rules and their use. A detailed treatment of the rule language and how to write rules is in Part II, Chapter 4.

Part I: Introduction to the Stanford Pascal Verifier

3. The Assertion Language

The assertion language is the language the programmer uses to document his programs. A documented program is a Pascal program containing assertions; assertions are required at certain points in programs, and are optional at other places. An assertion is a statement of relationships between program variables. It defines properties of computation states that must be true every time the position of the assertion is reached during a computation. For a theoretical discussion of assertions and the logic of Floyd-Hoare proofs, refer to [11, 12, 14].

The assertion language of the Stanford verifier permits logical statements within the quantifier-free first-order theories of arithmetic, Arrays, Records, and Pointers (i.e., the standard Pascal data types). Essentially this is the language of Pascal Boolean expressions extended in the following way:

- auxiliary user-defined predicate and function symbols are allowed
- priorities of the standard Pascal operators conform to mathematical conventions rather than Pascal
- special data structure terms have been introduced (see below).

There is not much of a theory of designing assertion languages at present. Assertion languages may well become program specification languages later on. We have tried to keep ours simple, adding new features only when the need for them is clear.

3.1 Kinds of assertion statements

Different kinds of assertions are allowed by the assertion language. We have introduced eight kinds of assertions to aid stating specifications. Four of these apply to the specification of procedures. In addition to ENTRY and EXIT assertions there are two others:

The INITIAL declaration is used to describe the values of a parameter before and after a procedure call. If procedure $p(x)$ adds 1 to x we cannot simply say $x>0$ $\{p(x)\}$ $x=x+1$. A convention denoting tense is needed. An INITIAL statement allows naming entry values, e.g., INITIAL $x=x0 \wedge x>0$ $\{p(x)\}$ $x=x0+1$.

The GLOBAL declaration permits the user to declare global variables of a procedure as formal parameters. One important application of this is in dealing with pointer parameters. If a procedure has a parameter of type ↑T, it is often necessary to declare the reference class (below) of all objects of type T as a global variable. This permits the verifier to keep track of any side-effects.

Other kinds of assertion statements are intended for use to avoid having to repeat inductive

assertions unnecessarily. The examples in Section 5 show the use of some kinds of assertions; a complete list of kinds of assertions and compulsory assertions is given in Part II, Chapter 1.

3.2 Data structure terms

The axiomatic theory of data structure terms has been introduced into the assertion language to define the semantics of assignment and selection operations on the Pascal structured types ARRAY, RECORD, and POINTER. For example, a data structure term of the form <A,[I],E> denotes the array obtained from A by placing E in the ith. position; <A,[I],E>[J] denotes the jth. element of <A,[I],E>.

We have similar terms denoting assignments to dereferenced pointers. For each pointer type declaration, TYPE T=↑T0, the verifier introduces a reference class, called *T0, of all elements of type T0. Pointers of type T are related to *T0 just as array indices are related to arrays. Example: The reference class resulting from X↑:=E is denoted by the term, <*T0,⊂X⊃,E>.

The ordinary first-order assertion language is extended to express the effects of data structure operations. The newly introduced functions are defined axiomatically.

3.2.1 Reference class identifiers

We introduce new individual variables called reference class identifiers into the assertion language. They have the form,

*<identifier> where <identifier> is any legal Pascal type identifier.

Reference classes are not types in Pascal (although the syntax for bounded reference classes appears in the early version of the Pascal specification). They are assertion language primitives and behave very much like unbounded arrays. We will define the type of *T to be *reference class of* T.

3.2.2 Functions and predicates on data structures

New function symbols corresponding to the Pascal selection, assignment, and new operations on complex data type variables are introduced:

Selection:	x[y] (array selection), r.f (record selection), D⊂q⊃ (pointer selection)
Assignment:	<x, [y], z> (array assignment), <r, .f, z> (record assignment), <D, ⊂q⊃, z> (pointer assignment)
Extension:	Duq

The new terms formed by composition of these new functions must obey the Pascal type compatibility requirements. Thus x[y] is legal only if x is of array type and y is of the correct index type. Similarly <x,⊂y⊃,z> is legal only if x is of type reference class and x,y,z have compatible types. To do this, the new functions have types. The type of x[y] is the type of elements of the array term x. The type of <x,[y],z> is the same as that of x. If the type of x is *reference class of* T, the type of x⊂y⊃ is T. The type of <x,⊂y⊃,z> is the same as that of x. The type of D∪x is the same (reference class) type as D. The types for record terms are defined analogously.

The definition of terms in the assertion language is extended to accomodate new terms created by the combination of reference class identifiers and the special functions. Assertion language terms are:

1. all Pascal variables
2. all terms obtained from 1. and the new functions by function composition restricted to compatible types.

The new terms are called data structure terms.

Reference predicate: Pointer_To(X,D) means X is a pointer to a member of the reference class D.

3.2.3 Axioms for data structure terms

The selection and assignment functions satisfy the following axioms (all the free variables are universally quantified):

Ax 1. Y=U → <X, [Y], Z>[U]=Z
Ax 2. Y≠U → <X, [Y], Z>[U]=X[U]
Ax 3. <X, .Y, Z>.Y = Z
Ax 4. <X, .Y, Z>.U = X.U where Y and U are distinct identifiers
Ax 5. Y=U → <X, ⊂Y⊃, Z>⊂U⊃=Z
Ax 6. Y≠U → <X, ⊂Y⊃, Z>⊂U⊃=X⊂U⊃

The extension function obeys three axioms:

Ax 7. D∪X∪Y = D∪Y∪X
Ax 8. X≠Y → (D∪X)⊂Y⊃=D⊂Y⊃
Ax 9. X≠Y → <D, ⊂Y⊃, Z>∪X=<D∪X, ⊂Y⊃, Z>

Similarly, the predicate Pointer_To(X, D) obeys the following axioms:

Ax 10. Pointer_To(NIL, D)
Ax 11. Pointer_To(X, D∪X)
Ax 12. Pointer_To(X, <D, ⊂Y⊃, E>) = Pointer_To(X, D)
Ax 13. X≠Y → (Pointer_To(X, D∪Y) = Pointer_To(X, D))

A formulation of most of these axioms as verifier rules is given in 4.5.

Other standard lemmas may be derived from these axioms. For example, <A, [I], A[I]> = A can be obtained as follows:

<A, [I], A[I]> = A if and only if (∀j) <A, [I], A[I]>[j] = A[j]

We prove by cases.
Suppose j≠I. Then, <A, [I], A[I]>[j] = A[j] from Ax 2.
Suppose j=I. Then, <A, [I], A[I]>[j] = A[I]=A[j] from Ax 1.
In both cases <A, [I], A[I]>[j] = A[j]. Therefore, (∀j) <A, [I], A[I]>[j] = A[j].

These axioms form a first-order theory of data structures. The terms of this theory represent finite sequences of operations on data structures. The theorems are logical formulas containing equalities and inequalities between data structure terms.

For example, we can show that the formula

K≠I ∧ L=J → <<A, [I], <A[I], [J], 2>>, [K], B>[I][L] = 2

is a theorem of this theory. By axiom 2,

K≠I → <<A, [I], <A[I], [J], 2>>, [K], B>[I][L] = <A, [I], <A[I], [J], 2>>[I][L].

Axiom 1 implies <A, [I], <A[I], [J], 2>>[I][L] = <A[I], [J], 2>[L],
and finally L=J → <A[I], [J], 2>[L] = 2.

In order to express many complicated properties of data structures we need to introduce auxiliary predicates. For example, if we have Pascal type definitions,

type T0 = ↑T;
T = *record* ...; Next: T0; ...

it may be necessary to make assertions about "reachability" between pointers, i.e., from pointer x one can reach pointer y by performing the Next operation finitely many times. We introduce auxiliary predicates and add the axioms (D ranges over terms of type *reference class of* T):

Reach(D, x, y) $=_{df}$ (∃j) Reachstep(D, x, y, j)
Reachstep(D, x, y, 0) $=_{df}$ (x=y)
Reachstep(D, x, y, j+1) $=_{df}$ (∃z) Reachstep(D, x, z, j) ∧ D⊂z⊃.Next=y

Axiomatizations of auxiliary concepts must be supplied by the programmer as rules (see Section 4 and examples, especially 5.7).

The semantics of Pascal array, record, and pointer operations can be defined by Floyd-Hoare style axioms in terms of the theory of data structures. The actual semantics used in the verifier is given in Appendix C.

4. The Rule Language

4.1 Backward rules

Backward rules express logical implications, G→F, and are stated, INFER F FROM G. The rulehandler component applies these rules in a depth first backwards chaining search for proofs. A rule will apply to a problem, A→B, if B is an instance of F. INFER will then try to prove A→G' where G' is the corresponding instance of G. The rulehandler does not attempt to deduce new rules from the given set.

Example: In 5.2 we formulate a property of the *gcd* function:

```
GCD4: INFER GCD(X,Y)=GCD(MOD(X,Y),Y) FROM Y>0;
```

Again, note that this rule will only be applied by the system if an instance of $gcd(x, y) = gcd(mod(x, y), y)$ occurs as a result to be proven during the proof.

4.2 Replacement rules

These express logical equivalences between atomic formulas, F↔G, and equalities between terms, F=G, and are stated in the form: REPLACE F BY G. Whenever an instance of F occurs in a VC the equality F=G is asserted. (Note that F is not replaced by G, rather the notation "replace" has historic reasons.)

Example: The following is used in 5.8.5:

```
CONSTANT NULL_SEQUENCE;
CON4: REPLACE CONCAT(X,NULL_SEQUENCE) BY X;
```

This rule asserts that $concat(x, null_sequence) = x$. Note, however, that this equality only becomes known to the prover if an instance of $concat(x, null_sequence)$ occurs during the proof.

4.3 Forward rules

Forward rules also express an implication G→F, but they differ from backward rules in the way they are used in proof searches. These rules are written: FROM G INFER F. Forward rules can be used to derive consequences from a set of known facts.

Example: The inference rule given in 4.1 can be rewritten as:

```
GCD4F: FROM Y>0 INFER GCD(X,Y)=GCD(MOD(X,Y),Y);
```

In this case the fact expressed by the rule would be known to the system as soon as a term $y>0$ becomes true during a proof.

4.4 Differences between rules

Different rules may express the same logical statement. For instance the equivalence of two formulas A and B can be stated in at least the following three ways:

```
REPLACE A BY B;
INFER A FROM B; INFER B FROM A;
FROM A INFER B; FROM B INFER A;
```

The reason for this is that rules not only express logical facts; they also contain information for the prover on how and when to use those facts. Part II, Chapter 4 explains how the different kinds of rules are used.

The application of a rule can be limited by the use of restricting expressions. Suppose we want to express the fact that $x*y>0$ if $x>0$ and $y>0$. We could write:

```
FROM X>0 ∧ Y>0 INFER X*Y>0;
```

This rule might, however, lead to very inefficient proofs. For each pair of terms known to be positive, the fact that their product is positive will be asserted. From $x>0 \wedge y>0$ we derive not only $x*y>0$ but also $x*x*y>0$, $x*y*y>0$, and so on. We can avoid this by adding a <u>whenever</u> expression to the rule:

```
WHENEVER X*Y FROM X>0 ∧ Y>0 INFER X*Y>0;
```

The restriction X*Y limits the application of this rule to those x and y whose product appears in the formula to be proved. Again, note that the use of restrictions is explained in Part II, Chapter 4.

4.5 Rules for data structure terms

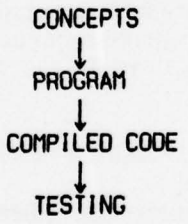
The axioms of the theory of data structures were given in 3.2.3. Below we give a set of rules expressing most of these axioms. The axioms omit the inequalities between all pairs of distinct record field identifiers. At the moment, only some of the theory is implemented by the simplifier and it is up to the user to include, in his rulefile, rules such as these to express any required data structure axioms:

```
ARR0: REPLACE <A, [I], E>[J] BY CASES I=J → E; I≠J → A[J] END;
REC0: REPLACE <A, .II, E>.JJ BY CASES II=JJ → E; II≠JJ → A.JJ END;
PNT0: REPLACE <A, ⊂I⊃, E>⊂J⊃ BY CASES I=J → E; I≠J → A⊂J⊃ END;
```

```
PNT1: REPLACE A∪⊂J⊃ WHERE I≠J BY A⊂J⊃;
PNT2: REPLACE <A, ⊂I⊃, E>∪J WHERE I≠J BY <A∪J, ⊂I⊃, E>;
PNT3: WHENEVER A∪X INFER POINTER_TO(X, A∪X);
PNT4: FROM POINTER_TO(X, A) INFER POINTER_TO(X, <A, ⊂Y⊃, E>);
PNT5: FROM POINTER_TO(X, A) INFER POINTER_TO(X, A∪Y);
PNT6: FROM ¬POINTER_TO(X, A∪Y) INFER ¬POINTER_TO(X, A);
PNT7: FROM TRUE INFER POINTER_TO(NIL, A);
PNT8: FROM POINTER_TO(X, A) ∧ ¬POINTER_TO(Y, A) INFER X≠Y;
```

5. Verification Examples

The paradigm employed in ordinary programming can roughly be described as follows: One starts out with some concepts that describe what the program is supposed to do and how it will do it. Such concepts may include arithmetical facts, properties of data structures, e.g., "array A is sorted", and procedures, e.g., "exchange the ith. and jth. elements of array A". These concepts are well enough understood that they are used to guide the human problem-solving activity that finally results in a program. Many attempts have been made to formalize this activity as an ordered sequence of steps, e.g., "requirements → code → documentation → testing", or by a "topdown" method. Despite these attempts, normal programming activity seems well described by the diagram,



In designing verifiable programs we advocate a completely different process. Again we start out with concepts. But before writing any code we develop a formal theory of the concepts involved. Often the concepts are already axiomatized (e.g., arithmetic) and one can use well known formal theories. In other cases (e.g., business applications) the necessary formalisms have to be developed from scratch. Hopefully this will change as more and more programs are verified and more theories for important programming concepts become available.

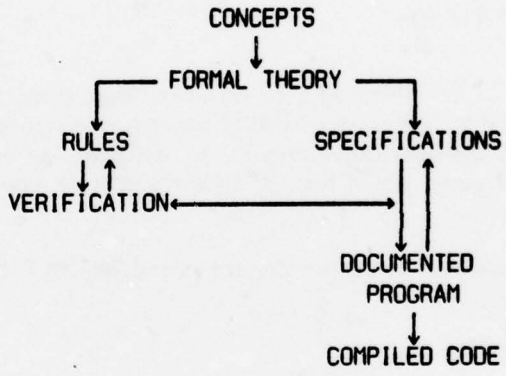
Using our formal theory of the initial concepts we can rephrase the original problem by precisely stating what the program is supposed to do within the formal theory. Now we are ready to embark on writing a program. This will be done with the theory in mind, and at any stage we may use documentation by inductive assertions (the assertions being formulas of the formal theory) to justify a particular piece of code. Additionally, some program statements, e.g., procedures and loops, must have formal inductive assertions stating their behavior - i.e., certain statements have a required documentation. This means in particular that each loop has an

associated invariant. The final product will be a program documented by precise formal statements.

In parallel with writing the program, the axiomatic theory defining the programming concepts must be expressed in a form accepted by the verifier, i.e., as "rules".

Finally, the program and the rules are submitted to the verifier. The result may or may not be a proof of the correctness of the program. If not, we have either written a wrong program or inadequate assertions, or the rules expressing the theory are insufficient for the system to find a proof. In each case we have to improve one of the above steps (specification, coding, rules) until a proof is established.

Graphically the verification paradigm for program development can be represented as follows:



5.1 First example: understanding VCs

This is a simple example in constructing documented programs and reading very simple VCs. We hope eventually to automate aids for analyzing VCs.

We begin by constructing a procedure that multiplies a given value parameter, Y, by a global value, N, and stores the result in X; its specifications are:

```
PROCEDURE CONSTMULT(VAR X:INTEGER; Y:INTEGER);
GLOBAL (N);
EXIT X=Y*N;
```

We could implement this by repeatedly adding Y to X in a loop; if we use Z to count the number of times the addition has been performed, we will expect X=Y*Z to be an invariant of the loop. This should be sufficient internal documentation. Finally, we will try calling CONSTMULT(X,N) to compute the square of N.

```
PASCAL
VAR N,Z:INTEGER;

PROCEDURE CONSTMULT(VAR X:INTEGER; Y:INTEGER);
GLOBAL (N);
EXIT X=Y*N;
VAR Z:INTEGER;
BEGIN
   X←0; Z←0;
   INVARIANT X=Y*Z
   WHILE Z≠N DO BEGIN X←X+Y;
                      Z←Z+1
                END
END;

EXIT Z=N*N;
BEGIN
   CONSTMULT(Z,N);
END.
```

For CONSTMULT to be consistent with its documentation, there are two VCs that must be proved. VCs tell us what theorems are needed to prove the correctness of paths in the program. The expressions in a VC are substitution instances of assertions and boolean tests in the program. We can recognize which paths are in the VC by the values of loop and conditional tests, and assertions appearing in the VC.

```
Unsimplified Verification Condition: CONSTMULT 1

0=Y*0 ∧
(X_0=Y*Z_0 ∧
 ¬(Z_0≠N)
 →
   X_0=Y*N)
```

This VC is of the form:

$$\text{INVARIANT}(0,Y,0) \wedge (\text{INVARIANT}(X_0,Y,Z_0) \wedge \neg\text{LOOPTEST}(Z_0,N) \rightarrow \text{EXIT}(X_0,Y,N)$$

It implies the consistency of two paths:

(i) The path from the entry to the loop before it is executed: the initial values of X, Y, and Z must satisfy the invariant, and since these values are, X=Z=0, this requires 0=Y*0.

(ii) The path from the loop to the exit: Since X and Z are variables of the loop, their final values may differ from their initial values, so VCGEN has given these final values the new names X_0 and Z_0.

```
Unsimplified Verification Condition: CONSTMULT 2

(X=Y*Z ∧
 Z≠N
 →
   X+Y=Y*(Z+1))
```

This VC is of the form,

$$\text{INVARIANT}(X,Y,Z) \wedge \text{LOOPTEST}(Z,N) \rightarrow \text{INVARIANT}(X+Y,Y,Z+1).$$

It corresponds to the path around the loop, and implies that X*Y=Z is an invariant. To prove it, the prover will need the distributive law of arithmetic, which may be expressed by a rule as follows:

```
RULEFILE(DISTRIBUTIVITY)
DIST: REPLACE A*(B+C) BY A*B+A*C;
```

It should be emphasized that such arithmetical rules can sometimes lead the prover into deducing many irrelevant facts; for this rule to have the desired effect, the verifier parameter SUMMATCH must be turned on (see Part II, Section 4.2.13).

Finally, proof of the procedure call depends on the VC,

```
Unsimplified Verification Condition: MAIN 1

(Z_0=CONSTMULT_X(Z,N,N) ∧
 Z_0=N*N
 →
   Z_0=N*N).
```

This is trivially true, but it is instructive to note what VCGEN is doing in constructing the VC. It is of the form,

```
Z_0=FUNCTION(<initial values of all parameters>) ∧
CONSTMULTEXIT(Z_0,N)
→
  EXIT(Z_0,N).
```

Z_0 is the final value of the actual VAR parameter Z (note this is the outer Z). This VC states that the result of the procedure call (i.e., the EXIT assertion of CONSTMULT instantiated to the final values of its actual parameters) may be assumed in proving the EXIT to the main program. Also, a function is constructed for each VAR parameter that maps the initial values of all parameters (including the globals) into the final value of that VAR parameter; VCGEN appends the formal parameter to the procedure name to make a unique function name (in this example, CONSTMULT_X). This reflects the semantics of procedure call in [13].

5.2 Concepts, documentation and verification

As a next example we will verify a simple greatest common divisor (*gcd*) program. The concept of the *gcd* is of course well known and we can base our documentation on the standard mathematical properties by using the following lemmas for non-negative x and y:

$gcd(x, 0) = x$
$gcd(x, x) = x$
$gcd(x, y) = gcd(y, x)$
$gcd(mod(x, y), y) = gcd(x, y)$ *if* $y>0$
$mod(x, y) < x$

The program uses these properties by repeatedly replacing one of the values x or y by $mod(x, y)$.

```
PASCAL

FUNCTION MOD(I,J:INTEGER):INTEGER;
ENTRY I≥0 ∧ J>0;
EXIT MOD≥0;
EXTERNAL;

FUNCTION G(X0,Y0:INTEGER):INTEGER;
ENTRY X0>0 ∧ Y0>0;
EXIT G=GCD(X0,Y0);
VAR X,Y,R:INTEGER;
BEGIN
   X←X0; Y←Y0;
   REPEAT R←MOD(X,Y);
          X←Y;
          Y←R
   UNTIL Y=0
   INVARIANT GCD(X0,Y0)=GCD(X,Y) ∧ X>0 ∧ Y≥0;
   G←X
END;.
```

The invariant for the REPEAT-loop follows immediately from our basic idea of replacing one argument of $gcd(x, y)$ by $mod(x, y)$ and thereby not changing the value of *gcd*.

The next step towards a verification is to express the facts about *gcd* mentioned above in a form acceptable to the verifier. In the rule language these facts can be expressed in various ways: one can use forward or backward rules or any combination thereof. In the first case the prover would deduce all terms equal to a term x as soon as it sees x. Going backwards the prover would try to prove an equality only if it is needed. There is no general rule telling us which is better, each method has its own advantages and disadvantages. Let us specify the properties of *gcd* with forward rules.

```
RULEFILE(GCD)

GCD1: REPLACE GCD(X,0) BY X;
GCD2: REPLACE GCD(X,X) BY X;
GCD3: REPLACE GCD(X,Y) BY GCD(Y,X);
GCD4: REPLACE GCD(X,Y) WHERE Y>0 BY GCD(MOD(X,Y),Y);
```

Rewriting these rules as backward rules leads to the following rulefile, which is not sufficient for the proof of all the verification conditions:

```
RULEFILE(GCD)

GCD1: INFER GCD(X,0)=X;
GCD2: INFER GCD(X,X)=X;
GCD3: INFER GCD(X,Y)=GCD(Y,X);
GCD4: INFER GCD(X,Y)=GCD(MOD(X,Y),Y) FROM Y>0;
```

Two verification conditions require commutativity (rule GCD3); these two formulas cannot be proved with this set of rules. The reason is that the backward rule GCD3 is only applied if the system tries to prove a formula that matches the pattern of the INFER clause. If we change the rule GCD3 into

```
GCD3: INFER GCD(X,Y)=Z FROM GCD(Y,X)=Z;
```

we greatly increase the number of possible matches; in fact, using this modified rule, one can verify the *gcd* program.

5.3 A hard invariant

The following example demonstrates that finding a suitable invariant is not always a simple task. We want to emphasize, however, that this example is not typical of problems arising in practice. In general we have some intuitive idea of what a loop is supposed to do and this will lead us to finding the right invariant (in fact we ought to be able to write the invariant before we write the code for the loop). In this example we find ourselves in the position of verifying a rather tricky program and finding its loop invariant requires understanding the trick. The program is an iterative version of McCarthy's 91-function [21]. This function is recursively defined as

$$f(x) = \textit{if } x>100 \textit{ then } x-10 \textit{ else } f(f(x+11))$$

It can be shown that this recursive function computes

$$f(x) = \textit{if } x>100 \textit{ then } x-10 \textit{ else } 91$$

Now we want to show that the following program computes the same function:

```
PASCAL
LABEL 1;
VAR X,Y1,Y2,Z:INTEGER;
ENTRY TRUE;
EXIT (X>100 ∧ Z=X-10) ∨ (X<101 ∧ Z=91);
BEGIN
   Y1←X; Y2←1;
1:
   ASSERT ????;
   IF Y1>100 THEN
               IF NOT(Y2=1) THEN BEGIN
                                    Y1←Y1-10;
                                    Y2←Y2-1;
                                    GOTO 1
                                 END
                            ELSE Z←Y1-10
            ELSE BEGIN
                    Y1←Y1+11;
                    Y2←Y2+1;
                    GOTO 1
                 END
END.
```

The entry and exit assertions simply state that the program computes the same function as the recursive 91-function. The difficult part is to find a suitable invariant at ????. The key, of course, is to first understand the operation of the program.

Each time label 1 is reached the program starts computing f. There are two possible cases depending on Y1. If the initial value of Y1>100, the program terminates immediately. In the other case function f calls itself recursively, i.e., f(f(Y1+11)). The program computes the inner call to f by jumping back to label 1. But in addition, it has to be recorded that upon completion of this computation, the outer call has to be computed. This is done by incrementing the variable Y2; thus Y2 tells us how many outer calls remain to be evaluated whenever we reach label 1.

Suppose at a given point in time all remaining outer calls will take the Y1>100 branch. Then each time Y1 will be decreased by 10 and Z will become Y1-10*Y2. Since in this case Z has to be 91, we propose the invariant Y1-10*Y2=91. But this turns out to be too strong. It might be the case that all but one of the outer calls are evaluated and we arrive at label 1 in a situation where Y2=1 and Y1<101. In this case the loop will take the Y1≤100 branch and new recursive calls have to be evaluated. Thus the invariant will only be Y1-10*Y2<92. This is still insufficient, but the remaining details are fairly easy to find. First, we have to take care of the case where X>100, i.e., the program terminates immediately. Second, we will need the fact that throughout the loop Y2 is positive, so we have to add the conjunct Y2>0. Altogether we get the invariant:

$$((X>100 \wedge Y2=1 \wedge Y1=X) \vee (X<101 \wedge Y1-10*Y2<92)) \wedge Y2>0$$

The following is a terminal session showing the verification of this program. Note that the prover has to do some non-trivial reasoning to prove MAIN 4. The boldface characters were typed by the user.

```
r verify

Hi there, welcome to the Pascal Verifier.
 Version (VCG 4, SIMP 24.) (October 4 ...)
Type 'HELP;' for help

>read itf91.ver;
Reading file: ITF91.PAS[EX,VER]
SYNTAX SCAN COMPLETE.
PROGRAM PARSED.
CPU SECONDS:0.389

>printvc;
>

Unsimplified Verification Condition: MAIN 1

(0<Y2 ∧
 (100<X ∧
 Y2=1 ∧
 Y1=X ∨
 X<101 ∧
 Y1-10*Y2<92) ∧
 100<Y1 ∧
 ¬(Y2=1)
 →
    (100<X ∧
    Y2-1=1 ∧
    Y1-10=X ∨
    X<101 ∧
    (Y1-10)-10*(Y2-1)<92) ∧
    0<Y2-1)


Unsimplified Verification Condition: MAIN 2

(100<X ∧
1=1 ∧
X=X ∨
X<101 ∧
X-10*1<92) ∧
0<1


Unsimplified Verification Condition: MAIN 3

(0<Y2 ∧
 (100<X ∧
 Y2=1 ∧
 Y1=X ∨
 X<101 ∧
```

```
 Y1-10*Y2<92) ∧
 100<Y1 ∧
 ¬¬(Y2=1)
 →
   100<X ∧
   Y1-10=X-10 ∨
   X<101 ∧
   Y1-10=91)


Unsimplified Verification Condition: MAIN 4

(0<Y2 ∧
 (100<X ∧
 Y2=1 ∧
 Y1=X ∨
 X<101 ∧
 Y1-10*Y2<92) ∧
 ¬(100<Y1)
 →
   (100<X ∧
   Y2+1=1 ∧
   Y1+11=X ∨
   X<101 ∧
   (Y1+11)-10*(Y2+1)<92) ∧
   0<Y2+1)
.
>simplify;
>

Simplified Verification Condition: MAIN 1

TRUE

Simplified Verification Condition: MAIN 2

TRUE


Simplified Verification Condition: MAIN 3

TRUE


Simplified Verification Condition: MAIN 4

TRUE
```

5.4 Defining concepts to document a program

The next program we will verify returns the maximum value of an array.

To formalize the concept of the maximum of an array we define the predicate $maxof(x, a, l, r)$ to be true if x is the maximum of the array elements $a[i]$ with $l \le i \le r$. We can give a formal definition of *maxof* as:

$$maxof(x, a, l, r) =_{df} (\forall i)\ (l \le i \le r \rightarrow a[i] \le x) \wedge (\exists j)\ (l \le j \le r \wedge a[j] = x)$$

From this definition the following lemmas are immediate:

$$maxof(a[l], a, l, l)$$
$$maxof(x, a, l, r) \wedge a[r+1] \le x \rightarrow maxof(x, a, l, r+1)$$
$$maxof(x, a, l, r) \wedge a[r+1] > x \rightarrow maxof(a[r+1], a, l, r+1)$$

These lemmas may be written directly as backward rules without any changes of propositional structure because they are all simple implications between conjunctions of atomic formulas. The rules below, however, are weaker than these lemmas. They are sufficient for the verification of this implementation of *max* because the array is scanned from 1 to N.

The full input submitted to the verifier for this problem is given below. Pascal Plus permits arrays in inner blocks to be dimensioned using VAR variables and this is the reason for the enclosing procedure DUMMY. (Note ":=" and "←" are both accepted as notation for assignment.)

```
RULEFILE(MAX)

M1: INFER MAXOF(A[1],A,1,1);
M2: INFER MAXOF(X,A,1,I)     FROM I≥2 ∧ A[I]≤X ∧ MAXOF(X,A,1,I-1);
M3: INFER MAXOF(A[I],A,1,I) FROM I≥2 ∧ A[I]>X ∧ MAXOF(X,A,1,I-1);

PASCAL
VAR N:INTEGER;

PROCEDURE DUMMY;
EXIT TRUE;
TYPE NARRAY=ARRAY[1:N] OF INTEGER;

FUNCTION MAX(A:NARRAY):INTEGER;
GLOBAL (N);
ENTRY N>0;
EXIT MAXOF(MAX,A,1,N);
VAR TEMP,I:INTEGER;
BEGIN
   TEMP:=A[1];
   FOR I:=2 to N
   INVARIANT MAXOF(TEMP,A,1,I-1)
   DO
      IF(A[I]>TEMP) THEN TEMP:=A[I];
   MAX:=TEMP
```

```
END;

BEGIN END;.
```

It is instructive to look at the unsimplified verification conditions. At this stage the properties of *maxof* declared in the rulefile have not been applied.

```
Unsimplified Verification Condition: MAX 1

(0<N ∧
 2≤N
 →
   MAXOF(A[1],A,1,2-1) ∧
   (MAXOF(TEMP_0,A,1,(N+1)-1)
    →
      MAXOF(TEMP_0,A,1,N)))


Unsimplified Verification Condition: MAX 2

(0<N ∧
 N<2
 →
   MAXOF(A[1],A,1,N))


Unsimplified Verification Condition: MAX 3

(I≤N ∧
 2≤I ∧
 MAXOF(TEMP,A,1,I-1) ∧
 TEMP<A[I]
 →
   MAXOF(A[I],A,1,(I+1)-1))


Unsimplified Verification Condition: MAX 4

(I≤N ∧
 2≤I ∧
 MAXOF(TEMP,A,1,I-1) ∧
 ¬(TEMP<A[I])
 →
   MAXOF(TEMP,A,1,(I+1)-1))
```

The verifier partitions the paths of a program in a particular way and each VC corresponds to one of these paths. MAX 1 corresponds to the path ENTRY → enter FOR-loop ∧ exit FOR-loop

→ EXIT; TEMP_0 is the final value of TEMP on leaving the loop. MAX 2 corresponds to the path ENTRY → bypass FOR-loop → EXIT. MAX 3 and MAX 4 correspond to the two different paths around the loop.

In practice, the initial rulefile is usually inadequate for the proof of all VCs. In this case inspection of the unproven (but simplified) VCs will often suggest new rules or modifications. These are then added to the rulefile and run in the verifier. This procedure is then repeated until all VCs are proved.

5.5 Specifications for sorting

This Bubble sort example is documented by standard sorting concepts. Each concept has a simple first-order definition (except permutation, see [12, 26]). For example,

ORDERED(A, L, R) means array A is ordered in the range [L, R]:

$$ordered(a\ l, r) =_{df} (\forall i)\ (l \le i \wedge i < r \rightarrow A[i] \le A[i+1]).$$

PARTITION(A, L, I, R) means that each element of A in [L, I] is smaller than each element of A in [I+1, R]:

$$partition(a, l, i, r) =_{df} (\forall j, k)\ (l \le j < i \wedge i \le k \le r \rightarrow A[j] \le A[k]).$$

Rules defining sorting concepts, including permutation, are given in [5]. The rules state not only standard axioms satisfied by the concepts, e.g., transitivity of permutation, but also how the concepts are related when operations are performed on arrays. Here is an example from [5]:

```
ORD6a: INFER ORDERED(<A,[P],X>,L,R) FROM ORDERED(A,L,R) ∧ L<P ∧ P<R ∧ X≤A[P+1]
             ∧ X≥A[P-1];
```

Rule ORD6a states conditions under which the array obtained from A by placing X in A[P] is ordered.

The rules can be shown correct by proving them from the first-order definitions. The sorting concepts may be used to document many different sorting algorithms, and the same defining set of rules can be used for verification [5] (rules for the theory of data structures are also needed).

```
PASCAL
VAR N:INTEGER;

PROCEDURE DUMMY;
EXIT TRUE;
TYPE NARRAY=ARRAY[1:N] OF INTEGER;

PROCEDURE SORT(VAR A:NARRAY);
GLOBAL (N);
INITIAL A=A0;
ENTRY N≥1;
EXIT PERMUTATION(A,A0) ∧ ORDERED(A,1,N);
VAR I,J,TEMP:INTEGER;
BEGIN
   I:=1; J:=1;
   FOR I:=1 TO N-1
   INVARIANT PERMUTATION(A,A0) ∧ ORDERED(A,N-I+2,N) ∧
             PARTITION(A,1,N-I+1,N)
   DO
      FOR J:=1 TO N-I
      INVARIANT PERMUTATION(A,A0) ∧ ORDERED(A,N-I+2,N) ∧
                ISBIGGER(A[J],A,1,J-1) ∧ PARTITION(A,1,N-I+1,N)
      DO
         IF A[J]>A[J+1] THEN BEGIN
                                TEMP:=A[J];
                                A[J]:=A[J+1];
                                A[J+1]:=TEMP
                             END
END;

BEGIN END;.
```

5.6 A pointer example

The procedure below has a side-effect. It changes the contents of the cell referenced by its X parameter by manipulating Y. The problem is to verify this. The type declaration, PNTR, introduces the reference class *CELL of all cells referenced by pointers of type PNTR. *CELL is a variable of the computation of SIDEFFECT although it cannot be mentioned in the code. It must therefore be declared as a GLOBAL parameter of SIDEFFECT, and indeed as a VARiable GLOBAL.

```
PASCAL
TYPE PNTR = ↑CELL;
     CELL = RECORD CAR:INTEGER END;

PROCEDURE SIDEFFECT(VAR Y:PNTR; X:PNTR);
GLOBAL (VAR #CELL);
ENTRY X↑.CAR = 1;
EXIT  X↑.CAR = 2;
BEGIN
   Y:=X;
   Y↑.CAR:=2
END;.
```

The single verification condition for procedure SIDEFFECT is

```
(#CELL⊂X⊃.CAR=1 ∧
 POINTER_TO(Y,#CELL) ∧
 POINTER_TO(X,#CELL) ∧
 #CELL_0=<#CELL,
          ⊂X⊃,
          <#CELL⊂X⊃,.CAR,2>>
→
   #CELL_0⊂X⊃.CAR=2)
```

The identifier *CELL_0 refers to the reference class after the operation Y↑.CAR:=2 which changes one of the cells in *CELL (namely the one pointed to by Y). So the relationship between them is

*CELL_0 = <*CELL, ⊂Y⊃, <*CELL⊂Y⊃, .CAR, 2>>.

The assignment of the value of X to Y makes this equivalent to the form that appears in the VC. The VC is proved using rules for reference classes given in Section 4.5.

5.7 Verification of Pascal list structure operations

List structures are usually implemented in Pascal by means of pointers and records. Verification of programs that operate on lists requires introducing higher level concepts analogously to the sorting concepts for sorting operations on arrays. List operations are defined in terms of operations on reference classes.

The procedure INSERT in the example below inserts a new word into a *loopfree* list. To prove that INSERT preserves *loopfreeness* we use the Reach concept introduced in 3.2.3. The predicate Reach(D,x,y) is true if by refering to the NEXT field repeatedly, starting at x, one can reach y; i.e., the sequence, x, D⊂x⊃.Next, D⊂D⊂x⊃.Next⊃.Next, . . . in the reference class D contains the pointer y. This implies that there are no loops between x and y.

```
PASCAL
TYPE REF  = ↑WORD;
     WORD = RECORD COUNT:INTEGER; NEXT:REF END;

PROCEDURE INSERT(ROOT,Y,SENTINEL:REF);
GLOBAL (VAR #WORD);
ENTRY REACH(#WORD,ROOT,Y) ∧ REACH(#WORD,Y,SENTINEL) ∧
      Y≠SENTINEL ∧ Y≠NIL;
EXIT REACH(#WORD,ROOT,SENTINEL);
VAR Z:REF;
BEGIN
   NEW(Z);
   Z↑.NEXT←Y↑.NEXT;
   Y↑.NEXT←Z
END;.
```

The entry assertion implies that the list from ROOT to SENTINEL is *loopfree* and Y is a pointer to a word in the list. The procedure inserts a new member of the list between Y and its successor. The exit assertion implies that the result is still *loopfree*. This property of INSERT is easily verified using the rules for data structures and some rules defining Reach.

Here are three examples of rules defining Reach:

```
R1: INFER REACH(D,X,Y) FROM REACH(D,X,Z) ∧ REACH(D,Z,Y);
R2: REPLACE REACH(<D,cX>.COUNT,E>,Y,Z) BY REACH(D,Y,Z);
R3: INFER REACH(<D,cY>.NEXT,Z>,X,W)
       FROM REACH(D,X,Y) ∧ REACH(D,Z,W) ∧ ¬INBETWEEN(D,Y,Z,W);
```

Rule R1 is implied by the transitivity of Reach. Rule R2 states that operations on the COUNT field, i.e., X↑.COUNT ← E, preserve *loopfreeness*. Finally, rule R3 states some conditions under which the assignment, Y↑.NEXT ← Z, preserves *loopfreeness* between X and W. We can justify the rules by proving them from the recursive definition of Reach given in 3.2.3. It is a challenging exercise to construct axiomatizations of Reach that are complete in the sense that all satisfying interpretations are isomorphic to linear lists.

Finally, suppose we reverse the last two statements of INSERT:

```
BEGIN
   NEW(Z);
   Y↑.NEXT←Z;
   Z↑.NEXT←Y↑.NEXT
END;.
```

The result of the attempted verification is:

```
Simplified Verification Condition: INSERT 1

(REACH(#WORD,ROOT,Y) ∧
 REACH(#WORD,Y,SENTINEL) ∧
 Y≠SENTINEL ∧
 Y≠NIL ∧
 POINTER_TO(ROOT,#WORD) ∧
 POINTER_TO(Y,#WORD) ∧
 POINTER_TO(SENTINEL,#WORD) ∧
 POINTER_TO(Z,#WORD) ∧
 ¬POINTER_TO(Z_0,#WORD) ∧
 #WORD_1=<#WORD∪Z_0,
          ⊂Y⊃,
          <#WORD⊂Y⊃,.NEXT,Z_0>> ∧
 #WORD_0=<#WORD_1,
          ⊂Z_0⊃,
          <#WORD_1⊂Z_0⊃,.NEXT,Z_0>>
→
  REACH(#WORD_0,ROOT,SENTINEL))
```

The identifier Z_0 represents the new value of Z; *WORD_0, and *WORD_1 are reference classes resulting from operations performed by INSERT. The conclusion of the VC is that *WORD_0 is *loopfree* between ROOT and SENTINEL. But if we look at the expression for *WORD_0 in the premise (this expression results from simplifications obtained from applying the data structure rules) we see that the NEXT field of Z_0 is Z_0, clearly a loop. As the expression for *WORD_1 shows that the NEXT field of Y is pointing to Z_0, so this loop is between ROOT and SENTINEL, the desired result is false.

5.8 A larger example

We now present a verification of a simple parser. Here we have available the well developed theory of context free grammars to assist us in documenting the parser. This theory provides us with the necessary concepts. Using user defined predicates and rules, these concepts can then be defined for use in the verification.

5.8.1 Theory

We will briefly review the theory underlying the proof. A context free grammar is a tuple $\langle T, NT, P, \{s\}\rangle$ where T and NT are the sets of terminal and nonterminal symbols, respectively. The character s is a distinguished start symbol in NT and P is a relation over $NT \times (T \cup NT)^*$. The sets T, NT, and P are all finite. Whenever $\langle l, r\rangle$ is in P, then r is of finite length.

The relation "=>" is defined over $(T \cup NT)^* \times (T \cup NT)^*$ as follows:

$$\langle u.t.v,\ u.w.v\rangle \in \Rightarrow \textit{iff } \langle t, w\rangle \in P.$$

We use periods to denote the concatenation of sequences over $(T \cup NT)^*$. The relation "=>*" is defined to be the reflexive and transitive closure of =>.

The goal of the parser is to determine whether or not a given sequence over T^* is in the language generated by the grammar; i.e., whether $s \Rightarrow^* u$ for the input u. To express the theory in our assertion language we introduce the following two predicates:

$$isprod(t, w) \text{ iff } \langle t, w \rangle \in P$$

$$isderiv(x, v) \text{ iff } x \in NT \wedge \langle x, v \rangle \in \Rightarrow^*$$

From the definition of $\Rightarrow^*$ one immediately gets two lemmas

$$isderiv(x, x)$$

$$(isderiv(x, u.t.v) \wedge isprod(t, w)) \supset isderiv(x, u.w.v)$$

5.8.2 The parsing algorithm

The parsing algorithm is standard (see [1], p177); we use a stack automaton and generate a top down leftmost derivation of the input string. More precisely, we start with a stack containing the start symbol s. Then we repeatedly take the top element t from the stack and if it is a nonterminal symbol we push a w on the stack such that $isprod(t, w)$. Otherwise, if t is a terminal symbol and it conforms with the first symbol in the input, we skip this first symbol. If none of these cases applies we report an error.

5.8.3 Implementation

First we decide upon the representation of the sets T, NT, and P in our program. The set T will be an enumerated type called *token* and the set NT will also be as type *nonterm*. We introduce a special type for $T \cup NT$; this will be a record called *item*. Note that this could well be a variant record; our system does not support variant records as such but does provide union types.

Sequences, that is elements from T^* and $(T \cup NT)^*$, are represented as files; i.e., T^* corresponds to *token_sequence*, a *file of token* and $(T \cup NT)^*$ corresponds to *t_nt_sequence* which is a *file of item*. Note that for an actual implementation we would have to change *t_nt_sequence* to some type that can be represented in memory (e.g., linked lists). However, for the presentation of this example we will use files; a change in the data structure would not affect the overall structure of the verification.

The representation of P is left undefined at this point; we assume the existence of an external procedure *isrhs* which given t will return a w such that $isprod(t, w)$ holds. The decision of what production is to be applied next is hidden inside *isrhs* and not specified further. To allow a reasonable implementation of *isrhs* we pass as an additional parameter the next character of the input, as lookahead. Thus our parser can deterministically recognize any LL(1) grammar.

Three external procedures *empty*, *push* and *top* implement a stack of *item*. Note, that *push* pushes a whole sequence on the stack rather than a single element.

An external procedure *error* is used to issue error messages.

We distinguish between single elements of $(T \cup NT)$ and the sequence of length one of $(T \cup NT)^*$; the function *make_sequence* takes an $x \in (T \cup NT)$ and converts it into $\langle x \rangle \in (T \cup NT)^*$.

5.8.4 Specifications

As might be obvious by now, we cannot prove that the parser will accept every legal input string, because we have not made strong enough assumptions about *isrhs*.

Instead we will prove the following statement: if the parser terminates and does not issue an error message, then the input string is in the language generated by the grammar.

This might seem to be a very weak statement; it is, however, a good illustration to demonstrate the difference between robustness, reliability, and correctness. With a suitable implementation of *isrhs* the parser will reliably parse any legal input string; an implementation of the procedure *error* can guarantee a reasonable recovery from syntax errors, thus making the program robust. In the case where the parser terminates without an error message, the program proof will guarantee a correct parsing of the input regardless of the actual implementations of *error* and *isrhs*.

In writing the assertions for this program we use the following functions:

imbed	maps a sequence over T^* into a sequence over $(T \cup NT)^*$
concat	concatenates two sequences
append	appends a single element to a sequence
con1	places a single element in front of a sequence.

The invariant of the main loop states that the input read so far concatenated with the contents of the stack is derivable from the start symbol. There is no magic in finding this invariant; it corresponds closely to the induction hypotheses of the formal proof that each context free grammar is accepted by a non-deterministic push down automaton [1],(p177).

To be able to formulate the invariant we include a virtual variable *source_read* which at any point contains the portion of the input read so far.

5.8.5 Rules

The rules necessary for the proof of the parser can be divided into two parts. In the first part, we have rules describing the properties of *isprod* and *isderiv*. Furthermore we have to specify properties of the auxiliary functions used, i.e., *append*, *concat*, *con1*, *imbed*, *make_sequence*.

The rules ISD1 and ISD2 formulate the two lemmas for *isderiv* mentioned above. The rules IMB1 through IMB6 express that *imbed* distributes over *make_sequence*, *con1* etc. IMB7 and IMB8 define *imbed* for a single element, i.e., this is mapped into one component of the record. In the second part, we give rules that express trivial facts about sequences.

The final rulefile is:

```
RULEFILE(PARSER)

CONSTANT NULL_SEQUENCE,INFO1;

ISD1: INFER ISDERIV(X,MAKE_SEQUENCE(X));
ISD2: INFER ISDERIV(X,CONCAT(Z,CONCAT(R,T))) FROM
        ISDERIV(X,CONCAT(APPEND(Z,L),T)) ∧
        ISPROD(L,R);

IMB1: REPLACE IMBED(MAKE_SEQUENCE(X)) BY MAKE_SEQUENCE(IMBED(X));
IMB2: REPLACE IMBED(CON1(X,Y)) BY CON1(IMBED(X),IMBED(Y));
IMB3: REPLACE CONCAT(IMBED(X),IMBED(Y)) BY IMBED(CONCAT(X,Y));
IMB4: REPLACE IMBED(CONCAT(X,Y)) BY CONCAT(IMBED(X),IMBED(Y));
IMB5: REPLACE APPEND(IMBED(X),IMBED(Y)) BY IMBED(APPEND(X,Y));
IMB6: REPLACE IMBED(APPEND(X,Y)) BY APPEND(IMBED(X),IMBED(Y));
IMB7: WHENEVER IMBED(X) FROM TRUE INFER X=IMBED(X).INFO1;
IMB8: WHENEVER X.INFO1 FROM TRUE INFER X=IMBED(X.INFO1);


NS1:  WHENEVER EMPTY(X) FROM EMPTY(X) INFER X=NULL_SEQUENCE;
NS2:  FROM TRUE INFER IMBED(NULL_SEQUENCE)=NULL_SEQUENCE;
NS1A: WHENEVER EMPTY(X) FROM ¬EMPTY(X) INFER X≠NULL_SEQUENCE;
MS1:  REPLACE CONCAT(MAKE_SEQUENCE(X),Y) BY CON1(X,Y);
APP1: REPLACE APPEND(NULL_SEQUENCE,X) BY MAKE_SEQUENCE(X);
FR1:  WHENEVER FIRST(X) FROM TRUE INFER CON1(FIRST(X),REST(X))=X;
CON1: REPLACE CONCAT(X,CON1(U,V)) BY CONCAT(APPEND(X,U),V);
CON2: REPLACE CONCAT(APPEND(X,U),V) BY CONCAT(X,CON1(U,V));
CON3: REPLACE CON1(X,NULL_SEQUENCE) BY MAKE_SEQUENCE(X);
CON4: REPLACE CONCAT(X,NULL_SEQUENCE) BY X;
CON5: REPLACE CONCAT(NULL_SEQUENCE,X) BY X;
EOF:  REPLACE EOF(X) BY EMPTY(X);
```

We start out by attempting to verify the following version of the parser:

```
PASCAL
TYPE
    TOKEN           = (EMPT,
                       IDENT,
                       NUMBER,
                       PLUS_SYMBOL,
                       AND_MANY_MORE);

    NONTERM         = (START_SYMBOL, AND_SOME_MORE);

    TOKEN_SEQUENCE  = FILE OF TOKEN;
    TERM_OR_NOT     = (NONTERMINAL, TERMINAL);

    ITEM            = RECORD
                        KIND:  TERM_OR_NOT;
                        INFO1: TOKEN;
                        INFO2: NONTERM
                      END;

    T_NT_SEQUENCE   = FILE OF ITEM;

VAR
    SOURCE, SOURCE_READ   : TOKEN_SEQUENCE;
    R, STACK              : T_NT_SEQUENCE;
    STRT, T               : ITEM;
    LOOK                  : TOKEN;
    DONE                  : BOOLEAN;


PROCEDURE ERROR;
ENTRY TRUE;
EXIT ERROR_MSG(1);
EXTERNAL;

PROCEDURE ISRHS(VAR R: T_NT_SEQUENCE; T: ITEM; L: TOKEN);
ENTRY TRUE;
EXIT ISPROD(T,R);
EXTERNAL;

% Procedures implementing a stack %
FUNCTION EMPTY(ST: T_NT_SEQUENCE): BOOLEAN;
ENTRY TRUE;
EXIT TRUE;
EXTERNAL;

% Return the top of the parsing stack, pop this element %
PROCEDURE TOP(VAR X: ITEM);
GLOBAL (STACK);
INITIAL STACK=S0;
ENTRY TRUE;
EXIT (~EMPTY(S0) → S0=CON1(X,STACK)) ∧ (EMPTY(S0) → ERROR_MSG(1));
EXTERNAL;
```

```
% Push x on the parsing stack: note that we push whole sequences
  rather than single elements. %
PROCEDURE PUSH(X: T_NT_SEQUENCE);
GLOBAL (STACK);
INITIAL STACK=S0;
ENTRY TRUE;
EXIT STACK=CONCAT(X,S0);
EXTERNAL;

% This function converts an element into a sequence of one element. %
FUNCTION MAKE_SEQUENCE(X: ITEM): T_NT_SEQUENCE;
ENTRY TRUE;
EXIT TRUE;
EXTERNAL;


% Main program %
INITIAL SOURCE=SOURCE0;
ENTRY ¬ERROR_MSG(1) ∧ EMPTY(STACK) ∧
      EMPTY(SOURCE_READ) ∧ ¬EMPTY(SOURCE);
EXIT ¬ERROR_MSG(1) → ISDERIV(STRT,IMBED(SOURCE0));
BEGIN
   STRT.KIND←NONTERMINAL; STRT.INFO2←START_SYMBOL;
   PUSH(MAKE_SEQUENCE(STRT));
   READ(SOURCE,LOOK);
   INVARIANT ¬ERROR_MSG(1) →
             (SOURCE0=CONCAT(SOURCE_READ,CON1(LOOK,SOURCE)) ∧
             ISDERIV(STRT,CONCAT(IMBED(SOURCE_READ),STACK)))
   WHILE NOT EOF(SOURCE) DO
      BEGIN
         TOP(T);
         IF T.KIND=TERMINAL THEN
            IF T.INFO1≠LOOK THEN ERROR ELSE
               BEGIN
                  WRITE(SOURCE_READ,LOOK);   % virtual %
                  READ(SOURCE,LOOK)
               END
         ELSE BEGIN
                 ISRHS(R,T,LOOK);
                 PUSH(R)
              END
      END;
   IF NOT EMPTY(STACK) THEN ERROR;
END.
```

An attempt to verify this program succeeds in establishing the truth of 4 out of the 5 verification conditions generated. The following VC is the only one which does not simplify to *TRUE*:

```
(¬ERROR_MSG(1) ∧
EMPTY(STACK) ∧
EMPTY(SOURCE_READ) ∧
¬EMPTY(SOURCE) ∧
SOURCE=SOURCE0 ∧
```

```
STRT_3=<STRT,.KIND,NONTERMINAL> ∧
STRT_2=<STRT_3,.INFO2,START_SYMBOL> ∧
STACK_7=PUSH_STACK(MAKE_SEQUENCE(STRT_2),SOURCE_READ) ∧
STACK_7=MAKE_SEQUENCE(STRT_2) ∧
SOURCE_4=READ_F_F(SOURCE0,LOOK) ∧
LOOK_4=READ_X_X(SOURCE0,LOOK) ∧
LOOK_4=FIRST(SOURCE0) ∧
SOURCE_4=REST(SOURCE0) ∧
EMPTY(SOURCE_3) ∧
SOURCE0=APPEND(SOURCE_READ_2,LOOK_3) ∧
ISDERIV(STRT_2,CONCAT(IMBED(SOURCE_READ_2),STACK_6)) ∧
EMPTY(STACK_6)
→
  ISDERIV(STRT_2,IMBED(SOURCE0)))
```

One way to prove this formula is to show that

imbed(*source*0) = *concat*(*imbed*(*source_read*_2), *stack*_6).

Given that *source*_3 and *stack*_6 are both empty this means showing

imbed(*append*(*source_read*_2, *look*_3)) = *imbed*(*source_read*_2).

But unfortunately, this VC is false; it cannot be proved from any set of consistent rules. Consequently, the VC reveals an error in our program.

Investigating further, we find that the unproved verification condition comes from the path which starts at the entry assertion of the main program, goes to the main loop, and then to the exit assertion of the main program. Looking at our program closely we find that in fact the main loop is not coded correctly. In the case where we read the last token from *source* into *look* the main loop will terminate. However, we haven't yet made the necessary reductions to derive the entire input string.

Having found this error we change the program to the following one:

```
% Main program %
INITIAL SOURCE=SOURCE0;
ENTRY ¬ERROR_MSG(1) ∧ EMPTY(STACK) ∧
      EMPTY(SOURCE_READ) ∧ ¬EMPTY(SOURCE);
EXIT ¬ERROR_MSG(1) → ISDERIV(STRT,IMBED(SOURCE0));
BEGIN
   STRT.KIND←NONTERMINAL; STRT.INFO2←START_SYMBOL;
   PUSH(MAKE_SEQUENCE(STRT));
   READ(SOURCE,LOOK);
   DONE←FALSE;
   INVARIANT ¬ERROR_MSG(1) →
             ((¬DONE → SOURCE0=CONCAT(SOURCE_READ,CON1(LOOK,SOURCE))) ∧
             ISDERIV(STRT,CONCAT(IMBED(SOURCE_READ),STACK)) ∧
             (DONE → (EMPTY(SOURCE) ∧ SOURCE0=SOURCE_READ)))
   WHILE NOT DONE DO
      BEGIN
```

```
            TOP(T);
            IF T.KIND=TERMINAL THEN
               IF T.INFO1≠LOOK THEN ERROR ELSE
                  BEGIN
                     WRITE(SOURCE_READ,LOOK);   % virtual %
                     IF EOF(SOURCE) THEN DONE←TRUE ELSE READ(SOURCE,LOOK)
                  END
            ELSE BEGIN
                     ISRHS(R,T,LOOK);
                     PUSH(R)
                  END
         END;
      IF NOT EMPTY(STACK) THEN ERROR;
   END.
```

This corrected program can be verified using the rulefile given above. To show that this proof is not at all trivial we include one of the unsimplified VCs:

```
(¬DONE ∧
 (¬ERROR_MSG(1)
  →
    (¬DONE
     →
       SOURCE0=CONCAT(SOURCE_READ,CON1(LOOK,SOURCE))) ∧
    ISDERIV(STRT,CONCAT(IMBED(SOURCE_READ),STACK)) ∧
    (DONE
     →
       EMPTY(SOURCE) ∧
       SOURCE0=SOURCE_READ)) ∧
 (EMPTY(STACK)
  →
    ERROR_MSG(1)) ∧
 (¬EMPTY(STACK)
  →
    STACK=CON1(T_0,STACK_2)) ∧
T_0=TOP_X(T,STACK) ∧
STACK_2=TOP_STACK(T,STACK) ∧
T_0.KIND=TERMINAL ∧
T_0.INFO1≠LOOK ∧
ERROR_MSG(1) ∧
¬ERROR_MSG(1)
→
  (¬DONE
   →
     SOURCE0=CONCAT(SOURCE_READ,CON1(LOOK,SOURCE))) ∧
  ISDERIV(STRT,CONCAT(IMBED(SOURCE_READ),STACK_2)) ∧
  (DONE
   →
     EMPTY(SOURCE) ∧
     SOURCE0=SOURCE_READ))
```

PART II

1. DIFFERENCES FROM STANDARD PASCAL

The verifier accepts most of the constructs of Pascal, modified in some cases for assisting verification. What follows is a list of the known differences between the language accepted by the verifier and "standard" Pascal as presented in Jensen and Wirth [15]; this list does not discuss the syntax or semantics of the rule language also accepted by the parser.

1.1 Comments

The scanner for all code ignores statements surrounded by percent (%) signs. Thus, comments may be added to code in this manner.

1.2 Program files

The Pascal code begins with the word PASCAL. The last character in the file should be a period (.). An end-of-file, except from the terminal, is accepted in lieu of a final period. A main program need not be present. Procedures must have a body, but it can be empty.

1.3 Procedure definitions

The GLOBAL, INITIAL, ENTRY, and EXIT statements (in that order) may follow a PROCEDURE or FUNCTION statement. The first three are optional; the last one must be there. For example:

```
PROCEDURE P(VAR X: INTEGER; Y: REAL);
GLOBAL (A; VAR Z);
        %Here the global Z may be changed by this procedure;
          the global A may be referenced by this procedure.%
INITIAL X=X0,Z=Z0;  %X0 and Y0 may appear only in assertions.%
ENTRY FOO(X,Y,A);
EXIT MUMBLE(X,X0,Y) ∧ BUMBLE(A,Z0);
TYPE .....
VAR .....
BEGIN ..... END;
```

Functions may not have an INITIAL statement.

In the outermost block of a program, the ENTRY and EXIT assertions appear immediately preceding the BEGIN that starts the block. The order is ENTRY then EXIT, or just EXIT. An INITIAL statement, if present, precedes the ENTRY/EXIT assertions. Thus:

```
PASCAL
TYPE .....
VAR .....
PROCEDURES AND FUNCTIONS .....

EXIT MUMBLE(A,B);
BEGIN
      %main block of the program%
END.
```

1.4 Assertions

The ASSERT, COMMENT, and ASSUME documentation statements have been added to Pascal for verification purposes.

```
ASSERT <formula>
COMMENT <formula>
ASSUME <formula>
```

The ASSERT statement breaks a proof into two separate verification conditions. The COMMENT statement does not cause a break, but adds an additional fact (which must be verified) to the verification condition. The ASSUME statement does not cause a break; it adds an additional assumption to the verification without requiring proof. For futher details see Appendix C.

Each repetitive statement requires an invariant to be specified. Thus:

```
INVARIANT <formula> WHILE ---- DO ----
FOR ---- INVARIANT <formula> DO ----
REPEAT ---- UNTIL ---- INVARIANT <formula>
```

1.5 Blocks

As in PASCAL, declarations must appear in the order LABEL, CONSTANT (or CONST), TYPE, VAR, functions and procedures. Unlike Pascal, you may have more than one CONST, TYPE, or VAR statement.

1.6 Types

A previously known integer or real variable identifier may appear as one of the array bounds for the purpose of defining an array type by subrange.

Variant records and sets have not been implemented. Functions and procedures may not be passed as parameters.

The type CHAR has been implemented, but only character constants one character long may appear in programs. Packed arrays are not implemented. The character delimiter is the single-quote e.g., 'a'.

A TYPE may not be redefined as another TYPE within its scope. It may be redefined as a constant, var, procedure, or function; however that makes the type invisible within the scope of that redefinition and will cause a syntax error if any attempt is made to reference vars of the redefined type.

1.7 Functions

There is very strict enforcement of rules to ensure that functions have no side-effects. The following are prohibited in functions (not procedures):

VAR parameters
The NEW statement
Calling procedures that change globals
Changing globals
READ, WRITE, and REWRITE statements

Note that a reference class is a global. Thus, assigning to any dereferenced pointer will cause an error.

1.8 Input/Ouput

The only I/O statements allowed are EOF, READ, REWRITE, and WRITE. EOF takes an entity of type FILE and returns *TRUE* or *FALSE*. READ and WRITE each take only two arguments; the first is a file and the second is an entity of the same base type as the file. Files may be declared in the usual manner; however, an entity of type FILE may appear in executable code only in the READ, WRITE, and REWRITE statements (or be passed as a parameter).

1.9 Global variables

Any global variable that can be changed in a procedure must appear in a GLOBAL statement for that procedure, in a list preceded by VAR. Any global variable that can be referenced in a procedure must appear in a GLOBAL statement for that procedure, either preceded by VAR or not. It will generally lead to VCs which are easier to prove if a global is not preceded by VAR unless required.

Any global variable that is referenced by a function must appear in the GLOBAL statement for that function. Functions may not have VAR global variables.

Globals passed as parameters to another procedure are checked to be in the appropriate GLOBAL list of the first procedure.

A variable in your program may not have the same name as a function, predicate, or record field. However, the same name may be used as a record field in two different types of records.

1.10 Virtual variables and Passive statements

The word VIRTUAL may precede the word VAR in a declaration or a procedure or function parameter definition. In addition, the word VIRTUAL may proceed a non-VAR parameter definition. VIRTUAL entities may appear in documentation (ENTRY, EXIT, ASSERT, COMMENT, ASSUME, PASSIVE) or they may be passed to other virtual entities. They may not be used elsewhere.

The PASSIVE statement has been added to permit assignment to virtual variables. It is merely an assignment statement preceded by the keyword PASSIVE. This is the only way in which a virtual variable may be assigned to.

1.11 Operator precedence

The precedence for operators appearing in documentation and rules is different than in Pascal. In particular, there are many more levels of precedence. The symbol "∨" is the lowest priority, then "∧", and so on, in what seems to be a natural ordering (the specific ordering is contained in the syntax charts). For this reason, the symbols used in documentation to represent the logical operators are different than the AND, OR, and NOT of Pascal. For this purpose, documentation is the formulas following INVARIANT, COMMENT, ASSERT, ASSUME, ENTRY, and EXIT.

A limited form of type checking is performed in all documentation statements noted above. A variable appearing within a statement must be declared and known; expressions must make sense (thus addition cannot be performed on a Boolean variable, for example). However, there is no

requirement that function and predicate names be known. Parameters to these functions and predicates are not checked. The exception to this is the PASSIVE statement, which must meet the same (stricter) type-checking requirements as the assignment statement.

As part of the no side-effect enforcement, the verification condition generator checks to ensure that the same data may not be passed to a procedure in two different ways. This situation is signalled by a syntax error.

1.12 Union types

The construct UNION has been added to replace variant records. UNION is a general type constructor which can be combined with other types in the same way as ARRAY and RECORD. There is a TAG function for determining the tag of a union variable, and there are selection and construction functions.

The UNION type declaration has the form

```
TYPE untype = UNION a1: t1; . . . ; an: tn END;
```

where the ti are types and the ai are constants of an enumerated type or integer subrange. If the ai are of an enumerated type, the type must have been declared previously, and each of its elements must appear once in the UNION declaration.

Assuming that u and u1 are variables of a union type untype (above) and x is a variable of one of the ti types, then the following operations are defined:

```
VAR u, u1: untype;
    x: ti;
```

SELECTION u:ai returns the ai component of u.

At any time, only one of the components of u exits. Selection of u:ai is an error if the tag of u is not ai.

TAG function TAG(u) returns one of the constants ai, the current tag.

CONSTRUCTORS untype:ai(x) returns a value of untype with tag ai.

As a consequence of the declaration of untype, separate constructor functions are defined for each of the ai. The constructor untype:ai takes values of type ti and converts them into values of the union type.

ASSIGNMENTS

```
u := u1;
u:ai := x;  valid only if TAG(u)=ai
u := untype:ai(x);
u := x;  implicitly applies construction
```

Assignment to a union variable of a value of the same type is always permitted. An assignment to a component of a union variable, as in the second statement, is permitted only if that component currently exists in u. In the third statement, u is set to the union value constructed from the value of x. The fourth statement is equivalent to the third one: the parser determines from the mismatch between the types of u and x, that the constructor untype:ai must applied.

Example: The data structure and basic operations of LISP defined in Pascal with union types.

```
PASCAL

TYPE TAGS = (A,D,N);
     LISP = ↑U;
     DTPR = RECORD
                 CAR: LISP;
                 CDR: LISP
            END;
     ATOM = RECORD
                 VALUE: LISP;
                 PLIST: LISP
            END;
     U    = UNION
                 D: DTPR;
                 A: ATOM;
                 N: INTEGER
            END;

PROCEDURE CONS(X,Y: LISP; VAR RESULT: LISP);
GLOBAL (VAR #U);
EXIT TAG(RESULT↑)=D ∧ RESULT↑:D.CAR=X ∧ RESULT↑:D.CDR=Y;
VAR CELL: DTPR;
BEGIN
   NEW(RESULT);
   CELL.CAR:=X;
   CELL.CDR:=Y;
   RESULT↑:=U:D(CELL)
END;

FUNCTION CAR(X: LISP): LISP;
GLOBAL (#U);
ENTRY TAG(X↑)=D;
EXIT TRUE;
BEGIN
   CAR:=X↑:D.CAR
END;
```

```
PROCEDURE PLUS(X,Y: LISP; VAR RESULT: LISP);
GLOBAL(VAR #U);
ENTRY TAG(X↑)=N ∧ TAG(Y↑)=N;
EXIT TAG(RESULT↑)=N ∧ RESULT↑:N=X↑:N+Y↑:N;
BEGIN
   NEW(RESULT);
   RESULT↑:=X↑:N+Y↑:N;
                % note implicit application of U:N() to
                  convert INTEGER to type U %
END;
```

2. USER COMMANDS

A system command consists of a command keyword, possibly followed by some arguments, and a terminating ";". The semicolon must always be present. Most command keywords can be abbreviated to an initial substring that identifies the command unambiguously.

There are four classes of commands:

(1) imperative commands, which call the various parts of the verifier:

   (a) READ, READVC and PRINTVC for reading (parsing) and writing files in user-readable format;
   (b) SIMPLIFY and RESIMPLIFY for calling the theorem prover;
   (c) DUMP commands and LOAD commands for writing and reading files in internal format;
   (d) DELRFILE, DELRULE for selective deletion of rulefiles and rules.

(2) commands that set system parameters: ALIAS, SET, RESET, OPENFILE, CLOSE.

(3) commands for obtaining some sort of information from the system: HELP, SHOW, STATUS.

(4) commands for system control: QUIT, LISP.

The following sections describe the command syntax informally; the formal syntax is given in Appendix A.

2.1 Imperative commands

Most of the imperative commands take a file name as an (optional) argument. The syntax of file names is exactly the same as at monitor level. Unless specified otherwise, the system will assume unit DSK: and the current default PPN (see also the ALIAS command). Some commands will assume default file names if parts of a file name are omitted. The defaults for file names are explained in the description of the individual commands. In order to override a default extension an empty extension can be forced by "."; e.g., FOO.[X,BAZ].

READ commands

A READ command parses source code (rulefiles, programs, VCs), i.e., input in external format. Input is read either from the keyboard or from a file. The system determines from the keyword (the first word in the file) what kind of data it is reading. It announces what it is doing, and gives the names of the VCs and rules. A READ command takes a file name as (optional) argument. If no argument is given, reading is done from the keyboard. The command READ is

used for parsing Pascal source code and rulefiles. The command READVC is for reading in VCs in external format. The command READ also knows about VC-files, i.e., the command "READ FOO.VC;" is equivalent to "READVC FOO.VC;". Examples:

READ;	parses source code typed in from the terminal;
READ FOO.BAR;	parses the file FOO.BAR;
READVC FOO;	parses the file FOO.VC, assuming that it contains VCs;
READ FOO.VC;	does exactly the same.

A READ command will not accept files with the extensions CRL, CVC, or CTB. Those files have to be read into the verifier using a LOAD command.

PRINTVC

The command PRINTVC prints out VCs, either to the terminal or to a file (or both). It takes a VC-specification and a file name as (optional) arguments. If no file name is given, printing is to the terminal. The syntax of the arguments is the same as for SIMPLIFY (see below for examples).

SIMPLIFY commands

The command SIMPLIFY calls the theorem prover. The prover attempts to simplify one or more VCs, using the rules that are currently loaded. The command takes a VC-specification, a file name and system parameter settings as (optional) arguments. If no VCs are specified, all current VCs are taken. If a file name is given, output is to that file; a copy can also be displayed on the terminal. If no file name is given, output is to terminal only. A list of system parameter settings (in parentheses) may appear either right after the command keyword or at the end (before the ";"). The command can be abbreviated to "S". Examples:

SIMPLIFY;	simplify all current VCs and display them on the terminal;
S (TRACE,PROOFDEPTH=5);	simplify all VCs with TRACE turned on and PROOFDEPTH set to 5;
SIMPL FOO 1(SHOWGOAL);	simplify VC 1 of FOO and display subgoals during the proof;
SIMPL TO FILE.EXT[A,FOO];	simplify current VCs and write simplified VCs onto file FILE.EXT[A,FOO];
SIMPL → FILE.EXT[A,FOO];	same as previous example, "→" may be used instead of "TO";
SIMPL MAIN COPY TO AAA;	simplify VCs of MAIN; write simplified VCs onto file AAA and display on terminal.

The RESIMPLIFY command takes the last VC returned by the simplifier and has another go at it. Sometimes this will have a beneficial effect.

DUMP commands

The group of DUMP commands includes the commands DUMP, DUMPVC, and DUMPRULE. A DUMP command produces a file containing VCs (DUMPVC), or rules (DUMPRULE), in

internal format so that at some later time they can be loaded directly using a LOAD command without requiring parsing. All DUMP commands use default file names. If no file name is given as argument, the default file name is VERIFY with an extension that depends on the command: CVC for VCs, CRL for rulefiles. These standard extensions are always used when a file in internal format is being created unless the user explicitly specifies a different (or empty) extension. The short command DUMP dumps both VCs and rules to appropriately named files; the argument to DUMP must be a simple file name without extension or PPN. It is advisable - and convenient - to make use of the default extension feature as the LOAD commands also know about them. Examples:

DUMPVC FOO;	write a file FOO.CVC containing current VCs;
DUMPVC FOO.BAR;	write a file FOO.BAR containing current VCs;
DUMPVC FOO.[P,PRO];	write a file FOO[P,PRO] containing current VCs;
DUMPRULE FOO;	write a file FOO.CRL containing current rules;
DUMP FOO;	write files FOO.CVC and FOO.CRL containing current VCs and rules, respectively.

If more than one rulefile exists, DUMPRULE will dump the one which was most recently parsed. A particular rulefile may be specified for dumping by giving its name as a second (optional) argument. Example:

DUMPRULE FILE, SRULES;	dump rulefile SRULES onto file FILE.

LOAD commands

The group of LOAD commands includes the commands LOAD, LOADVC, and LOADRULE. A LOAD command reads in a file which was previously created by a DUMP command. LOAD commands use the same conventions for naming files as the DUMP commands. If no file name is specified for LOAD, or if no extension is specified, all loadable files with the default name (VERIFY) and default extensions (CVC, CRL) will be used. The "long" commands load a file with an extension corresponding to their suffix. Examples:

LOADVC FOO;	loads the file FOO.CVC;
LOAD FOO.CVC;	does exactly the same;
LOADVC;	loads the file VERIFY.CVC;
LOAD FOO;	loads whichever (or all) of the files FOO.CVC and FOO.CRL exist.

DELETE commands

Rulefiles and rules can be deleted selectively by the commands

DELRFILE <list of rulefiles>;	for rulefiles, and
DELRULE <list of rule names>;	for rules.

The command DELRFILE without argument deletes all rules (so beware!).

2.2 Setting system parameters

ALIAS

The ALIAS command, like the monitor command, changes the project/programmer name (PPN) the verifier uses as default. It affects all file input and output to and from the verifier. Examples:

```
ALIAS VER,FOO;    changes the default PPN to [VER,FOO]:
ALIAS;            prints out the current default PPN.
```

SET, RESET

The user can set the values of various parameters that control the system. Parameter values can be changed either for the rest of the session with the SET/RESET commands ("sticky" changes), or temporarily in imperative commands. SET accepts as argument a list of parameters and values; if no parameter value is given to SET, it uses T (for *TRUE*). RESET sets parameters to their default values; this command accepts only a list of parameter names as argument. Examples:

```
SET TRACE, PROOFDEPTH=5;
RESET TRACE;
```

Sequences of SET command operands in parentheses may be included in a command string either after the keyword or at the end preceding ";" (for examples see the explanation of SIMPLIFY). The difference between setting parameters this way, or using SET/RESET, is that SET and RESET settings are permanent; settings given in a command string apply only for the duration of the command execution. If the same parameter name occurs twice, the first setting is overwritten. The type of parameter value expected depends on the parameter name. The following list gives user adjustable parameters with the type of their values:

```
natnum:   an integer greater than or equal to zero
bool:     a LISP flag: T or F (= NIL internally)
```

ASSERTDEPTH	natnum	maximum forward assertion depth;
CASEDEPTH	natnum	maximum depth of nesting of forward cases;
DEPTHTALK	bool	signal whenever a depth bound is reached;
PROOFDEPTH	natnum	maximum backward proof depth;
RULE	bool	enable rulehandler;
SHOWFACT	bool	enable assertion display during proof;
SHOWGOAL	bool	enable subgoal display during proof;
SHOWTEST	bool	enable display of tests made during proof;
SUMMATCH	bool	enable special sum matching: extra subspace instance;
TERMINAL	bool	if set, file output (from SIMPLIFY and PRINTVC) is also displayed on the terminal.
TRACE	bool	enable proof tracing;
TRACEFACT	bool	enable display of assertions made in trace output;
TRACEVC	bool	enable display of intermediate VCs during proof (works only if TRACE is set);

Current default values can be found using the SHOW command.

OPENFILE
The command OPENFILE opens a backup file. A backup file gets a copy of all the output from the system that is displayed on the terminal. It takes a file name as (optional) argument; the default file name is VERIFY.BKP. If the parameter (NOT) is included after the file name, output goes to the file only. A backup file can be closed using the command CLOSE. Example:

```
OPENFILE FOO (NOT);
```

Note that output cannot go to two different files simultaneously. Thus the backup file has to be closed before PRINTVC or SIMPLIFY can write onto other files, or another backup file can be opened. The system will notify you if this is necessary.

CLOSE
The command CLOSE closes a backup file. The command takes no argument.

2.3 Query commands

HELP
The HELP command provides information about various system features. It takes a keyword as argument. "HELP;" gives some general information about the verifier and pointers to further information. "HELP WHAT;" gives the list of topics for which help is available.

SHOW
The SHOW command displays the current values of system parameters. It takes a list of parameter names (separated by commas) as argument. If no arguments are given, SHOW displays the values of all parameters the system knows about.

STATUS
The STATUS command prints out a list of names of VCs, rulefiles and rules currently loaded. It takes no arguments.

2.4 System control

QUIT
The QUIT command is provided to allow one to exit gracefully from the verifier. "QUIT;" will return you to the monitor.

LISP
Typing "LISP;" to the system gets the user to the Maclisp toplevel. This command exists primarily for system maintenance and test; the uninitiated user should never need to use it. Once at LISP toplevel, evaluating (RESUME) will return control to the verifier command level.

Running the system

When loading the system, it will print out some more or less useful messages. As soon as the prompt character ">" appears, the system is ready to accept commands. The system tries to be fairly talkative; when executing a command it always prints out something. Thus, if the prompt character appears the system expects more input before it can execute the command (for example, the terminating ";" may have been omitted). All file manipulation is announced to the user, including full file names.

Error recovery

If for some reason or other the system ends up with a LISP error, evaluating (RECOVER) will return control to verifier command level. Typing <control> P will do exactly the same. If the error occurred in the simplifier, it will be reinitialized automatically.

3. DESCRIPTION OF THE SIMPLIFIER

3.1 Introduction

The prover has two components, a simplifier and a rulehandler (which is described in Part II, Chapter 4). The simplifier finds a normal form for any expression over the language consisting of individual variables, the usual boolean connectives, equality, the numerals, the arithmetic functions and predicates +, −, ≤, and <, the LISP constant and functions NIL, CAR, CDR and CONS, the functions ARRAYSTORE and ARRAYSELECT for storing into and selecting from arrays, the functions RECORDSTORE and RECORDSELECT for storing into and selecting from records, and uninterpreted function symbols. Individual variables range over the union of the reals, the set of arrays, the set of records, LISP list structure and the booleans *TRUE* and *FALSE*.

The simplifier is complete; that is, it simplifies every valid formula to *TRUE*. Thus it is also a decision procedure for the quantifier-free theory of reals, arrays, records, and list structure under the above functions and predicates.

The following are some examples of simplifications:

```
2 + 3*5
17

p ⊃ ¬ p
¬ p

x = f(x) ⊃ f(f(x)) = f(f(f(x)))
TRUE

x ≤ y ∧ y + d ≤ x ∧ 3*d ≥ 2*d ⊃ v[2*x - y] = v[x + d]
TRUE
```

The simplifier includes a number of cooperating special purpose provers, each a decision procedure for a particular quantifier-free theory. For instance, there is one prover for arithmetic, one for arrays, etc. Each prover has some modifications for use in the verifier; some of the modifications are temporary and reflect only the present version.

3.2 Prover for arithmetic

The axioms of this theory are:

$x + 0 = x$
$x + -x = 0$
$(x + y) + z = x + (y + z)$
$x + y = y + x$
$x \leq x$
$x \leq y \vee y \leq x$
$x \leq y \wedge y \leq x \supset x = y$
$x \leq y \wedge y \leq z \supset x \leq z$
$x \leq y \supset x + z \leq y + z$
$0 \neq 1$
$0 \leq 1$

The numerals 2, 3, . . . and ≥ are defined in terms of 0, 1, +, - and ≤ in the usual way. We also allow multiplication by integer constants; for instance, $2 * x$ abbreviates $x + x$.

The integers, rationals and reals are all models for these axioms. Any formula which is unsatisfiable over the rationals or reals can be shown unsatisfiable as a consequence of these axioms. Thus our simplifier is complete for the rationals or reals. It is not complete if the variables range over the integers, since there are unsatisfiable formulas, such as $x + x = 5$, which cannot be shown unsatisfiable as a consequence of the above axioms. The reason for the incompleteness is that determining the unsatisfiability of a conjunction of integer linear inequalities -- the integer linear programming problem -- is much harder in practice than determining the satisfiability of a conjunction of rational linear inequalities. This incompleteness is not as bad as it seems, since most formulas that arise in program verification do not depend on subtle properties of the integers.

In the present version, we have implemented one useful heuristic which makes the simplifier no longer sound for reals or rationals but which catches much of the incompleteness concerning integers. In addition to ≤, we allow < as a predicate symbol, but define $x < y$ to be $x + 1 \leq y$.

Notice in the description of the simplifier that multiplication is NOT mentioned although it appears in the examples. At the moment, we allow expressions such as $2 * x$ and there is some *ad hoc* code which tries to capture the more obvious properties of multiplication by constants, but the code makes no pretence of being complete. (The quantifier-free theory of integers under addition and multiplication is undecidable.)

3.3 Record prover

The record prover handles expressions involving storing into and selecting from records and record fields. The following axioms are implemented:

<r, .f, r.f> = r
<<r, .f, e1>, .f, e2> = <r, .f, e2>
<r, .f, e>.f = e
<r, .f, e>.g = r.g

With one exception, these are the axioms of the quantifier-free theory of records. The one axiom that is not implemented in the record prover concerns permutation of terms within data triples; that is, the axiom <<r, .f, e1>, .g, e2> = <<r, .g, e2>, .f, e1>. The reason for this omission is that this axiom can lead to combinatorial explosion. It appears to be rarely necessary in proofs and can be included as a rule if necessary.

The record prover can be turned on and off from LISP. Evaluating (RECORDPROVER) turns it on (and is the default); (NORECORDPROVER) turns it off.

3.4 Array prover

The array prover implements the following axioms for arrays:

<a, [i], a[i]> = a
<<a, [i], e1>, [i], e2> = <a, [i], e2>
<a, [i], e>[j] = (if i = j then e else a[j])

Again the axiom for permutations within data triples is missing. There are at the moment problems with the array prover in the verifier; because of an interface problem with the rulehandler, it is running much too slowly and requiring too much workspace. For this reason the arrayprover in the simplifier is temporarily defaulted to be off. It can be turned on in two ways from LISP. Evaluating (FASTARRAYPROVER) turns on a version which implements the first two axioms above plus the axiom <a, [i], e>[i] = e; it therefore lacks the axiom $i \neq j \supset$ <a, [i], e>[j] = a[j]. Evaluating (SLOWARRAYPROVER) turns on a version which implements the three axioms above. (NOARRAYPROVER) turns the array prover off and is the default.

3.5 List structure prover

Since Pascal does not have LISP list structure, the LISP special purpose prover has thus far not been turned on in the Pascal verifier.

3.6 Remarks

Complete descriptions of the various parts of the simplifiers and the component special provers appear in [22, 23, 24].

4. THE RULE LANGUAGE

4.1 Introduction to rules

We give an informal description of the rule language. A precise description of the syntax is given in Appendix A; this section is intended as a brief introduction to rules.

There are two types of rules: forward rules and backward rules. Roughly speaking, forward rules add new facts to the data base of the theorem prover as a consequence of old facts. Backward rules specify sets of subgoals which may be used in proving goals set up by the theorem prover. Some rules may cause "case splitting," which is the separation of a proof search into multiple contexts for the purpose of considering cases.

For each kind of rule, we give a brief description of the syntax, logical meaning, and semantics. The logical meaning specified is the "strongest" logical fact expressed by the rule. The semantics describe how this fact will be used by the theorem prover in proofs.

Certain conventions are used in the description below. Brackets in a syntactic description indicate an optional expression. A LITERAL is an atomic formula or a negated atomic formula. A TRIGGER-EXPRESSION is an expression which contains no propositional operators and which is not an individual variable. A REPLACEMENT-EXPRESSION is an expression which contains no propositional operators. An expression is an expression in the assertion language.

4.1.1 Backward rules

SYNTAX

infer A [from B] [whenever TR-1, TR-2, ...]

A: (the INFER clause) a conjunction of literals.
B: (the FROM clause) a conjunction of literals.
TR-i: a trigger-expression.

The WHENEVER and FROM clauses are optional. If there is no FROM clause, B is defaulted to *TRUE*. If there is a WHENEVER clause, it must have at least one trigger.

LOGICAL MEANING

$B \supset A$

SEMANTICS

A backward rule "applies" when the prover is trying to prove any of the literals in the INFER clause. If the FROM clause can be proved, the INFER clause is assumed to be proved. Multiple rules interact through standard subgoaling techniques. If A is the propositional constant *FALSE*, a contradiction will be derived if the FROM clause can be proved. Triggers in the WHENEVER clause restrict situations in which the rule will be applied to those in which instances of each trigger have occurred as subterms. Proof of the literals in the FROM clause proceeds from left to right.

EXAMPLES

```
infer A div B ≤ N from A≤N ∧ B≥1

infer Ordered(a,i,j) from Ordered(a,i,k) ∧ Ordered(a,k,j)


infer ISDERIV(X,MAKE_SEQUENCE(X))

infer ISDERIV(X,CONCAT(Z,CONCAT(R,T))) from
        ISDERIV(X,CONCAT(APPEND(Z,L),T)) ∧ ISPROD(L,R)

   % Two rules for verifying a context free parser:
        ISDERIV(X,Y) means there is a derivation from X to Y;
        ISPROD(L,R) means there is a production from L to R. %
```

4.1.2 Forward rules: FROM rules

SYNTAX

[whenever TR-1 , TR-2 , . . .] from B infer A
[whenever TR-1 , TR-2 , . . .] from B infer cases CASE-1 ; CASE-2 ; . . . end

A: (the INFER clause) a conjunction of literals.
B: (the FROM clause) a conjunction of literals.
TR-i: a trigger-expression.
CASE-i: a CASE (see below).

If there is a WHENEVER clause, it must have at least one trigger. A CASE must be one of the following two forms:

C → D or C

where C and D are conjunctions of literals. In the second case, D is defaulted to be *TRUE*. There must be at least one case in a CASES clause.

LOGICAL MEANING

B ⊃ A
B ⊃ [(C-1 ∧ D-1) ∨ (C-2 ∧ D-2) ∨ . . .]

SEMANTICS

When all of the WHENEVER triggers have been instantiated (there may be none), and when all of the literals in the FROM clause have become true, the INFER clause is asserted. If the INFER clause is a conjunction of literals, they are all asserted. If the INFER clause is a CASES construct, a case split is required. The actual split is delayed as long as possible (since a split is potentially expensive) but is done before any backward rules are applied. A case of the split may be eliminated during proof (but before the split is actually done) when its C-i formula (the formula to the left of the arrow) becomes false. If all but one of the cases are eliminated, no split is done; instead the remaining case is asserted immediately.

EXAMPLES

```
from P(S) infer ¬Q(X,Y,S)
   % When P(S) is true, Q(X,Y,S) is false, for all X and Y %

whenever A*B from A≥0 ∧ B≥0 infer A*B≥0

whenever X/Y from Y≠0 infer X=Y*(X/Y)

whenever MIN(I,J,K) from TRUE infer
  cases I≤J∧I≤K → I; J≤I∧J≤K → J; K≤I∧K≤J → K end
   % MIN(I,J,K)=I if I≤J and I≤K . . . %
```

4.1.3 Forward rules: REPLACE rules

SYNTAX

replace TR [where A] by RP
replace TR [where A] by cases C-1 → RP-1 ; C-2 → RP-2 ; . . . end

A: (the WHERE clause) a conjunction of literals.
C-i: a conjunction of literals.
TR: a trigger.
RP: a replacement.
RP-i: a replacement.

The WHERE clause is optional. If there is no WHERE clause, A is defaulted to *TRUE*. If there is a CASES clause, it must have at least one case.

LOGICAL MEANING

$A \supset (TR = RP)$
$A \supset [(C\text{-}1 \wedge TR = RP\text{-}1) \vee (C\text{-}2 \wedge TR = RP\text{-}2) \vee \ldots]$

SEMANTICS

When an instance of TR appears in the data base and the WHERE clause has become true, then do the action specified by the BY clause. If the BY clause is a replacement, then an equality (or equivalence) between TR and RP is asserted. If the BY clause is a CASES clause, a split is propagated. The two rules given in the syntax specification are equivalent to the following two FROM rules:

whenever TR from A infer TR = RP
whenever TR from A infer cases C-1 → TR = RP-1 ; C-2 → TR = RP-2 ; . . . end

EXAMPLES

```
replace X DIV 1 by X
   % Division by 1 %

replace A*B by B*A
   % Commutativity of multiplication.  This rule will not loop. %

replace <A,[I],E>[J] by cases I=J → E;  I≠J → A[J] end
   % Array data structure term simplification %

replace SIGN(X) by cases X≥0 → 1; X<0 → -1 end
   % Will cause a split if neither X≥0 nor X<0 can be shown %
```

4.1.4 Rulefiles

SYNTAX

RULEFILE(name)
[constant CS-1, CS-2, . . . ;]
[pattern PT-1, PT-2, . . . ;]
RN-1 : RULE-1 ; RN-2 : RULE-2 ; . . .

RULE-i: a rule.
RN-i: an identifier which will name the rule.
CS-i: an identifier which is to be a pattern constant.
PT-x: an identifier which is to be a pattern variable.

The CONSTANT and PATTERN specifications are optional. All identifiers appearing in the rulefile are assumed to be pattern variables except those used as function or predicate names, or as record field identifiers. These defaults can be overridden using the CONSTANT and PATTERN declarations.

SEMANTICS

A rulefile is a collection of rules. More than one rulefile can be active in the theorem prover at once. Each rule and each rulefile must have a unique name. Thus rules or rulefiles can be replaced by reading new rules or rulefiles with identical names; old rules or rulefiles with the same name are deleted. The order rules appear in the file is, more or less, the order in which they will be applied by the theorem prover.

EXAMPLE

```
rulefile(sample)

   constant NULL, EMPTY, CONST1, CONST2;
      % Declare various identifiers to be pattern constants %

   CONST:  from TRUE infer CONST1≠CONST2;
              % Assert that CONST1≠CONST2 to the data base %

   INEQ:   infer X>0 from X≠0 ∧ X≥0;
              % Rules like this may be required sometimes %

   APNULL: replace APPEND(NULL,X) by X;
              % NULL is a constant %

   GINFO:  replace G(X) where X.INFO=EMPTY by NULL;
              % INFO is a record field identifier,
                 and therefore not a pattern variable %
```

4.1.5 Switches and parameters

There are various parameters and switches for controlling the proof search and tracing. Several of these are depth bounds which allow the user to constrain the search in various ways. The SHOW switches are particularly useful for debugging rulefiles. In the default case, all trace switches are off. The SHOW command (see Section 2.3) can be used to determine the default settings of the depth bounds.

DEPTHTALK (switch) -- If this switch is set to true, the prover will print a message whenever it reaches a depth bound during search.

PROOFDEPTH (integer) -- This value is approximately the maximum depth of nesting of backward rules.

ASSERTDEPTH (integer) -- This value is approximately the maximum depth of nesting of assertions made by forward rules.

CASEDEPTH (integer) -- Approximately, the maximum number of forward case splits which will be allowed. All others will be ignored. This value does not include splits which are eliminated due to case reduction.

TRACE (switch) -- If this switch is set, a proof summary will be printed after simplification of a verification condition.

TRACEFACT (switch) -- If TRACE and TRACEFACT are both set, the proof summary will include a listing of facts asserted by forward rules.

TRACEVC (switch) -- An intermediate version ("presimplified") of the theorem to be proved will be printed. This version is the result of simplifying the formula in the presence of no rules. This output is useful for interpreting the TRACE results.

SUMMATCH (switch) -- If this switch is set, additional specific instances will be generated during matching of sums. The use of this switch is described in Section 4.2.13.

SHOWFACT (switch) -- This switch will cause the prover to display facts asserted by forward rules during simplification. Some of these facts may be asserted in inconsistent contexts, and may be false.

SHOWGOAL (switch) -- The theorem prover will display subgoals (from backward rules) generated during a proof if this switch is set. This feature is useful during development of rulefiles and assertions in programs. Some successful subgoals will not be displayed, because they are proved by TEST (see Section 4.2.7).

SHOWTEST (switch) -- The theorem prover will show all instantiated literals which are TESTed during proof if this switch is set (see Section 4.2.7).

4.1.6 An example

Here is a sample rulefile and two proofs which make use of it. The rulefile is not particularly efficient, though it does demonstrate several features of the rule language. The verification conditions come from an insertion sort program. The TRACE, TRACEFACT, and TRACEVC switches have been set. It took three seconds to prove INSERTSORT 2, and seven seconds to prove INSERTSORT 3. If only the rules ORD3 and ORD9 are included, proof of INSERTSORT 3 takes only two seconds.

```
RULEFILE(INSERT)
% Rulefile for insertion sort %

PERM1:   INFER  Permutation(I,I);
PERM2:   INFER  Permutation(Exchange(A,I,J),B)  FROM Permutation(A,B);
PERM3:   REPLACE <<P1,[P2],P1[P3]>,[P3],P4>  BY Exchange(<P1,[P2],P4>,P2,P3);

DATA1:   REPLACE <A,[J],X>[K]  WHERE K=J  BY X;
DATA2:   REPLACE <A,[J],A[J]>  BY A;

ORD1:    INFER  Ordered(K,I,J)  FROM I≥J;
ORD2:    INFER  Ordered(K,I,J)  FROM Ordered(K,I,L) ∧ Ordered(K,L,J);
ORD3:    INFER  Ordered(K,I,J)  FROM Ordered(K,L,M) ∧ L≤I ∧ J≤M;
ORD4:    INFER  Ordered(<K,[J],E>,I,L)  FROM I=J ∧ E≤K[I+1] ∧ Ordered(K,I+1,L);
ORD5:    INFER  Ordered(<A,[I],A[I-1]>,1,J)   FROM I<I ∧ I<J ∧ Ordered(A,1,J);
ORD6:    INFER  Ordered(<A,[I],A[I-1]>,1,J)   FROM I=J ∧ Ordered(A,1,J-1);
ORD7:    INFER  Ordered(<K,[J],E>,I,L)  FROM J=L ∧ K[L-1]≤E ∧ Ordered(K,I,L-1);
ORD8:    INFER  Ordered(A,1,I)  FROM Ordered(A,1,I-1) ∧ A[I]≥A[I-1];
ORD9:    INFER  Ordered(<K,[I],E>,J,L)  FROM J<I ∧ I<L ∧ Ordered(K,J,I-1) ∧
                    Ordered(K,I+1,L) ∧ K[I-1]≤E ∧ E≤K[I+1];

ARR:     INFER  K[L]≤K[M] FROM Ordered(K,I,J)∧ I≤L ∧ L<M ∧ M≤J;
```

Unsimplified Verification Condition: INSERTSORT 2

```
(ORDERED(K,1,J) ∧
 J≤N ∧
 X<K[I+1] ∧
 0<I ∧
 I<J ∧
 PERMUTATION(<K,[I],X>,K0) ∧
 0<I-1
 →
   (¬(K[I-1]≤X) ∧
    K_2=<K,[I-1+1],K[I-1]>
    →
      ORDERED(K_2,1,J) ∧
      J≤N ∧
      X<K_2[I-1+1] ∧
      0<I-1 ∧
      I-1<J ∧
      PERMUTATION(<K_2,[I-1],X>,K0)))
```

Presimplified Verification Condition: INSERTSORT 2

```
(ORDERED(K,1,J) ∧
 J≤N ∧
 X<K[1+I] ∧
 0<I ∧
 I<J ∧
 PERMUTATION(<K,[I],X>,K0) ∧
 2≤I ∧
 K[I-1]>X ∧
 K_2=<K,[I],K[I-1]>
 →
   ORDERED(K_2,1,J) ∧
   X<K_2[I] ∧
   PERMUTATION(<K_2,[I-1],X>,K0))
```

Simplified Verification Condition: INSERTSORT 2

TRUE

```
Proof summary for INSERTSORT 2.

 Assertions made by rules:
    DATA1: K_2[I]=K[I-1]
    PERM3: <K_2,[I-1],X>=EXCHANGE(<K,[I],X>,I,I-1)

 Top level goal ORDERED(K_2,1,J)
   Proof from backwards rule ORD5.

 Subgoal X<K_2[I]
   Proved without backwards rules.

 Top level goal PERMUTATION(<K_2,[I-1],X>,K0)
   Proof from backwards rule PERM2.


End of proof summary for INSERTSORT 2.


Unsimplified Verification Condition: INSERTSORT 3

(ORDERED(K,1,J) ∧
 J≤N ∧
 X<K[I+1] ∧
 0<I ∧
 I<J ∧
 PERMUTATION(<K,[I],X>,K0) ∧
 0<I-1
 →
   (K[I-1]≤X
    →
      (K_1=<K,[I-1+1],X>
       →
         ORDERED(K_1,1,(J+1)-1) ∧
         J+1≤N+1 ∧
         2≤J+1 ∧
         PERMUTATION(K_1,K0))))


Presimplified Verification Condition: INSERTSORT 3

(ORDERED(K,1,J) ∧
 J≤N ∧
 X<K[I+1] ∧
 0<I ∧
 I<J ∧
 PERMUTATION(<K,[I],X>,K0) ∧
 2≤I ∧
 K[I-1]≤X ∧
 K_1=<K,[I],X>
 →
```

```
   ORDERED(K_1,1,J))


Simplified Verification Condition: INSERTSORT 3

TRUE


Proof summary for INSERTSORT 3.

 Top level goal ORDERED(K_1,1,J)
   Proof from backwards rule ORD9.
   Subproofs:
     Subgoal ORDERED(K,1+1,J)
       Proof from backwards rule ORD3.

     Subgoal ORDERED(K,1,I-1)
       Proof from backwards rule ORD3.


End of proof summary for INSERTSORT 3.
```

4.2 Using the rule language

In this section we describe techniques for writing rules. The primary purpose of the rule language is to allow users of the verifier to supply lemmas to the theorem prover. By providing the necessary rules, the user can effectively extend the assertion language to include new concepts.

For example, let Ordered(A,i,j) mean that the array A is ordered in the interval [i,j]. By giving suitable rules, Ordered can be used in assertions in programs, and the theorem prover can be expected to prove a large variety of valid verification conditions involving Ordered.

Suppose we wish to express the following fact about Ordered:

(*) $(\forall A,i,j)\ (\text{Ordered}(A,i+1,j) \wedge i<j \wedge A[i] \leq A[i+1] \supset \text{Ordered}(A,i,j))$.

That is, if the array A is ordered in [i+1,j], and A[i] is not greater than the smallest element of A in the interval (namely, A[i+1]), then A is ordered in [i,j].

It would be nice if we only had to provide logical statements like (*), and proofs of valid verification conditions were forthcoming. However, the theorem prover does not have much

heuristic knowledge, and uses only the simplest methods to search for proofs. Therefore, when we provide a logical statement to the prover, we must tell it how to make use of that fact.

We start by distinguishing the two main types of rules. Then, a short description of the theorem proving algorithm is given. This provides the background for a more complete discussion of the differences between the two types of rules. Following this, some details are given about the ordering of proof search, to help the user improve the efficiency of his rules.

Pattern matching is then discussed. The matcher used in the rulehandler makes use of semantic knowledge in certain domains.

Several sections follow which describe various specific features of the rule language. Included among these features are rule schemata, a device for controlling application of rules through the use of the matcher, case splitting, for doing proof by considering cases, and semantic matching.

Finally, some general advice is given on efficiency considerations.

4.2.1 Forward and backward rules

Here is one way (*) can be expressed to the theorem prover:

R1: INFER Ordered(A,i,j) FROM i<j ∧ Ordered(A,i+1,j) ∧ A[i]≤A[i+1];

This has the effect of saying: "If you are trying to prove that A is ordered in [i,j], for any A, i, and j, then first prove that i<j, then Ordered(A,i+1,j), and finally, A[i]≤A[i+1]."

Here is another way of expressing (*):

R2: FROM Ordered(A,i+1,j) ∧ i<j ∧ A[i]≤A[i+1] INFER Ordered(A,i,j);

That is, for any A, i, and j, if you know that Ordered(A,i+1,j), i<j, and A[i]≤A[i+1] are all true, then you can assert Ordered(A,i,j). An equivalent way of writing R2 is:

R2A: FROM Ordered(A,i,j) ∧ i≤j ∧ A[i-1]≤A[i] INFER Ordered(A,i-1,j);

Rules like R1 are called BACKWARD rules; rules like R2 are called FORWARD rules. One way to think about backward rules is that they work backward: setting up subgoals from goals. Similarly, forward rules appear to work forward from assertions, generating new assertions. Backward rules may be compared to PLANNER consequent theorems; forward rules to antecedent theorems. Thus, though they may have the same logical meaning, they are applied differently in the search for a proof.

4.2.2 The theorem prover

This section will provide a rough idea of how the rulehandler of the theorem prover works.

In the theorem prover, many proofs can be accomplished without rules, since decision procedures for various theories (including equality and Presburger arithmetic) are built into the simplifier. The built in theories are described in Part II, Chapter 3.

The prover tries to prove theorems by deriving contradictions in a data base. Thus, if we are trying to prove $A \wedge B \supset C$, it asserts A and B, then asserts the negation of C, and finally tries to show that this data base describes an impossible situation. For example, suppose we want to prove $x=y \supset f(x)=f(y)$. We first assert $x=y$. Then we assert the negation of the conclusion: $f(x) \neq f(y)$. But by the properties of equality, these assertions cannot both be true, so the theorem is proved.

This method may be likened to the standard "Truthtable" method for simplifying propositional formulas, in that all possible assignments are considered for the propositional variables, "$x=y$" and "$f(x)=f(y)$." For each of these assignments, we must show either that the formula simplifies propositionally to $TRUE$, or that semantically the given assignment is impossible; that is, it describes a contradiction. Thus, in the example, there were four cases to consider. Three of them reduced to $TRUE$ propositionally. The fourth, assigning $TRUE$ to $x=y$ and $FALSE$ to $f(x)=f(y)$ resulted in a contradiction, eliminating that case from consideration. If we could not have eliminated this case semantically, the formula would not simplify to $TRUE$, because this case represented propositionally is $TRUE \supset FALSE$, or $FALSE$. Thus, data base contexts always represent conjunctions of literals; each literal is positive or negative depending on the truth-value assignment in the current (non-tautological) case.

4.2.3 Forward rules

Forward rules typically assert new facts to the data base as a consequence of old facts. For example, the rules FROM B∧A INFER D and FROM D INFER C are used to prove the verification condition A∧B ⊃ C in the following way: Initially, the data base is empty, and the rules are "waiting" for instances of B and D to be asserted. First, A is asserted to the data base, followed by B. After B is added, the first rule "fires" and waits now for an instance of A to be asserted (literals in a FROM clause are considered from left to right). A is already in the database, so the rule immediately continues and asserts an instantiated D to the data base. The state of the data base at this point may be represented by A∧B∧D.

After D is asserted, the second rule "fires" and asserts an instantiated version of C. Finally, ¬C is asserted from the verification condition, and a contradiction is now evident: A∧B∧D∧C∧¬C. Thus, A∧B ⊃ C has been proved using the two rules.

To prove A∧B⊃C∧D, multiple data base contexts would be used. First a contradiction would be derived from A∧B∧¬C; then a contradiction would be derived from A∧B∧¬D, giving the proof.

These two sub-proofs share a subset of the data base, A∧B. That is, in both cases, we have *TRUE* assigned to A and B. This means that forward rules triggered by A or B will "fire" once only, and the results will go into the common data base.

4.2.4 Case splits

It is possible for a rule to require splitting of the data base into multiple contexts for the purpose of considering cases. For example, the rule,

CAS: FROM A INFER CASES B; C END;

indicates that if A is true, then B∨C is true. That is, it indicates a disjunction between the elements of the CASES clause. This rule would be used to prove the data base, ¬B∧¬C∧A, inconsistent in the following manner: After A is asserted, the rule "fires" and indicates that a case split is required. Case splits are delayed as much as possible, to take advantage of sharing of common information in the multiple contexts. When the case split is done, two cases will be considered. To prove the theorem, a contradiction must be derived from both cases. In the first case, B is asserted to the data base, obtaining ¬B∧¬C∧A∧B, which is false. The other case, ¬B∧¬C∧A∧C, also simplifies to *FALSE*, resulting in a proof.

If more than one forward CASES rule applies, requiring multiple case splitting, the cases are nested, so the total number of cases considered will be the product of the numbers of cases propagated.

4.2.5 Backward rules

One way to think about a backward rule in this environment is to consider it as the contrapositive of a forward rule. Thus, the backward rule INFER C FROM D could be considered to be equivalent to FROM ¬C INFER ¬D. Now suppose we write a backward rule

C1: INFER A FROM B∧C;

The contrapositive of B∧C ⊃ A is ¬A ⊃ ¬B∨¬C. This could be written as the forward rule

C2: FROM ¬A INFER CASES ¬B; ¬C END;

From these two examples, it appears that all backward rules can be translated into equivalent forward rules. Is there any difference, in fact, between forward and backward rules? There is, and it will become apparent when we see how the system deals with more than one rule. Here are two backward rules for Ordered:

ORD1: INFER Ordered(a,i,j) FROM i<j ∧ Ordered(a,i,j-1) ∧ a[j-1]≤a[j];
ORD2: INFER Ordered(a,i,j) FROM i<j ∧ Ordered(a,i+1,j) ∧ a[i]≤a[i+1];

Suppose we are trying to prove Ordered(B,M,N). The proof will go as follows: We try to use rule ORD1 first, since it appears first. There are three cases to consider, corresponding to the three literals in the FROM clause of the rule. These cases are tried sequentially. If all of these cases do not simplify to *FALSE*, we abandon attempts at this rule and go on to consider the case split required by the rule ORD2, which also has three cases. Thus we will try at most six cases, at least two (one from each rule).

Each of the cases we try may have subcases, generated by rules that become applicable due to the new assertion made to the data base on that case. The process of applying backward rules and splitting in this manner is called SUBGOALING.

While backward-style splitting is more efficient than forward-style splitting, it is not COMPLETE, in that forward-style splitting may yield a proof in examples where backward-style splitting would not. The reader should be able to construct an example to illustrate this.

Thus, we distinguish between forward, complete splitting and backward, incomplete splitting. In a given data base, with forward splitting, each applicable rule multiplies the maximum number of cases considered; with backward splitting, each applicable rule adds to the maximum number of cases considered. Had forward complete splitting been used with ORD1 and ORD2, at most 9 cases would have been considered, rather than just 6. For this reason, it is desirable to use backward splitting (or subgoaling) whenever possible. To illustrate this: suppose there were 10 rules for Ordered similar to ORD1 and ORD2, each with three cases. If they were backward rules, we would consider at most 30 cases. Were they forward rules, we would have to consider some 3^{10} (that is, 59049) cases.

4.2.6 Ordering backward rules

In our proofs, splits are always delayed until all other assertions have been made to the data base. All backward rules are considered to propagate splits. This includes rules like INFER P FROM Q, which propagates a split with one case, and rules like INFER P, which propagates a split with no cases. The reader should be able to convince himself that the rules INFER P FROM Q and FROM ¬P INFER ¬Q are not equivalent for this reason: These rules are logically equivalent, but not heuristically equivalent because incomplete splitting is used in the backward rule.

When more than one backward rule applies, rules are tried in the order they appear in the rulefile, the data base of rules. By ordering rules carefully, the user can improve the speed of his proofs. Consider the following four rules:

N1: infer N(x,y) from P(x) ∧ Q(y);
N2: infer N(x,y) from S(x,y);
N3: infer N(x,y) from N(y,x);
N4: infer N(x,y) from N(C(x),C(y));

The "easier," non-recursive rules appear first. When trying to prove N(A,B), non-N subgoals

would be tried in the following order (assuming none were provable): P(A), S(A,B), P(B), S(B,A), P(C(B)), S(C(B),C(A)), P(C(A)), S(C(A),C(B)), P(C(C(A))), and so on. Q(x) is never tried, since the P(x) case always fails.

The rule N3 will not loop forever. When P(A) and S(A,B) fail, the rule sets up the goal N(B,A). After P(B) and S(B,A) fail (sub-subgoals from N1 and N2 on this subgoal), N3 applies again, setting up the goal N(A,B). But "setting up a goal" means denying a fact to the data base. Since ¬N(A,B) already exists in the data base, denying N(A,B) again produces no effect, so no new rules apply and the next subgoal, from N4, will be considered: N(C(B),C(A)). Infinite looping could arise from N4, however, unless there is a rule which expresses C(x)=x. In general, it may be extremely difficult, if not impossible to write non-recursive rules for certain concepts. For this reason, there are "depth-bounds" or cut-offs built into the rule mechanism to limit search.

Suppose the N rules had been ordered: N4, N1, N2, N3. Because we use a depth-first search paradigm, the rule N4 would be applied recursively until the depth bound was reached before any other subgoals were generated! Thus, if the depth bound were three, the first subgoal would be P(C(C(A))).

In fact, strict depth-first search is not used; the rulehandler uses a combination breadth-first depth-first search: All subgoals at a level are generated. If any of them can be proved without further backward rules, they will not be set up as subgoals. Thus, even with the bad order of rules, there would be no search in the proof of P(A)∧Q(A) ⊃ N(A,B).

Within a given rule, subgoals are tried in the order they appear in the FROM part of the rule. Thus, "easier" literals should appear first, since, if their proofs failed, further cases which may involve more extensive searching will not be tried. Considerations such as these would help the user decide how to order the subgoals in the rule N1.

4.2.7 Introduction to matching

Logically, rules are universally quantified statements, with quantification over all variables which appear. Thus, the rule

M1: FROM P(x)∧Q(x) INFER R(x);

represents the logical statement $\forall x[P(x) \wedge Q(x) \supset R(x)]$.

When a rule "fires," the effect internally is to make a copy of the rule with "constrained" quantification. For example, suppose we are trying to prove

P(A) ∧ Q(B) ∧ Q(C) ∧ A=B ⊃ R(B).

The first literal asserted to the data base is P(A). At this point, M1 fires, and waits for Q(A) to be asserted. One way of viewing this is that a new rule,

M1A: FROM Q(A) INFER R(A);

has been added to the system, and that to avoid duplication, M1 has now been constrained to fire with x distinct from A in this context. Q(A) is fully instantiated; in resolution terminology, it is a ground literal. This means we can TEST its validity directly, rather than merely waiting for "instances." Thus, if we had a forward rule

M2: FROM TRUE INFER Q(x);

this rule would apply during the test of Q(A), and M1A would assert R(A). Had the literals in M1's FROM clause been reversed, use of M2 would not have been possible since rules only apply to literals which appear in the data base (and thus are ground).

However, there is no rule M2 in our example, so testing Q(A) fails, and the rule M1A continues to wait. The next literal asserted to the data base from the theorem is Q(B). This does not fire any rules. P(C) is the next literal asserted. At this point M1 fires again, since C is distinct from A, and another virtual instance rule is created,

M1B: FROM Q(C) INFER R(C);

The data base is $P(A) \wedge Q(B) \wedge P(C)$. A=B is now asserted. At this point, the rule M1A fires, since Q(B)=Q(A) by the (built in) theory of equality, and R(A) is asserted. The data base is now

$$P(A) \wedge Q(B) \wedge P(C) \wedge A=B \wedge R(A).$$

Rule M1 is waiting for instances of P(x) where x is distinct from A,B, or C. Rule M1B is waiting for Q(C) to become true. Rule M1A has already fired for all of its possible instances (only one).

Finally, the denial of the conclusion of the theorem is asserted, ¬R(B). Since R(A) and A=B are both in the data base, a contradiction is indicated. Thus, we have proved the theorem using the rule M1.

We make several observations about this proof. Forward rules without CASES are always "waiting" on some literal pattern. If this literal pattern is not fully instantiated (for example, P(x) in M1), the prover will wait for instances to appear in the data base. On the other hand, if the literal is fully instantiated (for example Q(A) in M1A), the prover not only waits for the literal to appear, it also "tests" the literal for validity in the data base. This means that in each distinct context in the data base, the literal will be denied in an effort to obtain a contradiction. During the test of the literal, forward rules may be applied, resulting in proof of the literal in the given data base.

4.2.8 Ordering within rules

Suppose we want to prove $A>0 \wedge P(A+1) \supset Q(A+1)$, and we know $(\forall x)[x>0 \wedge p(x) \supset q(x)]$. There are two ways we could write forward rules to express this fact:

AR1: FROM x>0 ∧ P(x) INFER Q(x);
AR2: FROM P(x) ∧ x>0 INFER Q(x);

Consider using AR1. When A>0 is asserted, AR1 fires, and creates a virtual instantiated rule,

AR1A: FROM P(A) INFER Q(A);

in the data base, since x was bound to A when AR1 fired. With this rule, we cannot prove the theorem. Suppose we are using AR2 instead. After P(A+1) is asserted, the rule instantiates to

AR2A: FROM A+1>0 INFER Q(A+1);

Testing A+1>0 succeeds, since A>0 is in the data base, and thus is known to the built in Presburger arithmetic prover. Therefore, Q(A+1) is asserted, and the theorem is proved. This illustrates the fact that the order in which literals appear in a rule affects the ability of the system to obtain a proof.

This ordering constraint also holds for backward rules because cases are considered in the order they appear in the rule. Suppose we had the rules

ORD3: INFER Ordered(a,x,y) FROM x<y ∧ Ordered(a,x,z) ∧ Ordered(a,z,y)
ORD4: INFER Ordered(a,x,y) FROM x<y ∧ Ordered(a,z,y) ∧ Ordered(a,x,z)

Consider using ORD3 to prove, say, Ordered(B,I,J). The first subgoal is I<J, which is fully instantiated, and thus will be tested in the data base, and may require further backward rules for proof. If it is provable, then the rule waits for some z such that Ordered(B,I,z) exists in the data base. If it finds an instance, say where z=K, it sets up the further subgoal Ordered(B,K,J), which is fully instantiated and thus may use further backward rules for proof. Thus, Ordered(B,I,K) must actually appear in the data base, in order to provide an instance for z, while Ordered(B,K,J) need only be derivable from rules. Had we used ORD4, the situation would have been reversed. Thus, the two rules are not equivalent, and both may be required for some proofs.

This ordering constraint should not be viewed as a weakness of the rulehandler, since by giving all permutations, it could be circumvented. Indeed, it provides the user with a way of controlling proof search since he can predict which literals will be uninstantiated.

4.2.9 Rule schemata: Whenever and Replace

Suppose we desired to assert x*y≥0 whenever we saw a product, x*y and it was evident that x≥0 and y≥0. We could write

MUL1: FROM x≥0 ∧ y≥0 INFER x*y≥0;

Consider the effect of this rule. Whenever an assertion is made in the data base of the form E≥0

for any expression E, both literals in the FROM clause will match. Thus, for every pair E and F where both E≥0 and F≥0 (possibly E=F, of course), the rule will fire, asserting E*F≥0. This adds a new inequality assertion to the data base, and so the rule will match many times more. In the end, many useless facts will get asserted and much prover time will be wasted, since the rule matches indiscriminately.

We can remedy this by using a device called a RULE SCHEMA, which allows us to give a "trigger pattern." The rule

MUL2: WHENEVER x*y FROM x≥0 ∧ y≥0 INFER x*y≥0;

says that whenever a product, A*B, appears in the data base, an instantiated version of MUL2 will appear:

MUL2A: FROM A≥0 ∧ B≥0 INFER A*B≥0;

Here, all literals are instantiated, so the validity of the FROM literals can be tested. Further, the rule applies only to products which actually appear in the data base. Thus, adding the WHENEVER clause weakens the rule by restricting its application so it makes assertions only about products which appear in the data base. However, the WHENEVER clause also strengthens the rule by causing the FROM literals to be fully instantiated, and thus subject to testing in the data base. That is, the WHENEVER rule, MUL2 would prove

A≥0 ∧ B≥0 ⊃ (A+1)*B≥0

while MUL1 would not. While they have different heuristic meanings, logically, MUL2 and MUL1 express the same fact.

One very common application of WHENEVER is asserting equalities between terms. For example,

GCD1: WHENEVER GCD(x,y) FROM x MOD y = 0 INFER GCD(x,y)=y;

Another way of writing this rule is

GCD2: REPLACE GCD(x,y) WHERE x MOD y = 0 BY y;

This rule is semantically equivalent to GCD1. The "REPLACE" syntax is used for historical reasons; in fact, there is no actual rewriting or replacement -- an equality is asserted. Thus, REPLACE rules may be viewed as statements of directional equalities.

Because of the structure of the data base, rules like

TWIST: REPLACE F(x,y) BY F(y,x);

will cause no looping. Similarly, replacement rules can be provided for both directions of an equality:

ASC1: REPLACE F(x,F(y,z)) BY F(F(x,y),z);
ASC2: REPLACE F(F(x,y),z) BY F(x,F(y,z));

WHENEVER clauses may include more than one trigger pattern. All triggers must match before the instantiated rule will appear. Triggers are expressions which contain no propositional operators and are not individual variables.

4.2.10 Levels of proof

Thus far, we have seen that there are two levels of interaction between instantiated literals in rules and the data base. A literal in a rule is a FINAL literal if it occurs in the FROM clause of a backward rule or in the INFER clause of a forward rule. Final literals are those literals which, when instantiated, can get asserted "permanently" in a data base context, which may be the branch of a split. Since these literals become part of the data base, they can cause other rules to be applied, and further splits to be generated. Thus they have the same status as literals which actually occur in the theorem to be proved.

Literals which are not final are called TRANSITION literals. In general, the prover waits for transition literals to become true before "firing" a rule. When the rulehandler is attempting to establish validity of an instantiated transition literal, it will test that literal in the data base at various times. During testing, forward rules which don't split may be applied, as well as knowledge from the built in theories. Presently, there is also the restriction that when a transition literal is being tested, nested tests will not be done; that is, they will fail. Thus, the following rules will not work together as expected:

TR1: FROM P(x) ∧ Q(F(x)) INFER R(x);
TR2: REPLACE F(x) WHERE S(x) BY G(x);

The prover uses a process called FIND to locate instantiations for uninstantiated literals. In general, this means that a literal must be found in the data base which matches the pattern literal in the rule. In the case of equalities, the process is slightly more powerful. If both sides of the pattern equality are uninstantiated, an actual matching equality must be found in the data base. Otherwise, when only one side of the equality is uninstantiated, the prover will wait for an instance of this side of the equality to become equivalent to the value in the data base corresponding to the instantiated side of the equality pattern.

All uninstantiated literals are proved with FIND. Thus, if ATOM(A) is asserted,

ATOM: FROM ATOM(x) INFER x≠CONS(y,z);

will cause the prover to wait for an instance of CONS(y,z) to become equivalent to A. If such an instance appears, a contradiction will be propagated.

Uninstantiated literals should, of course, not be single pattern variables.

4.2.11 Forward cases

In order to increase efficiency of proofs, a case elimination mechanism has been built into the forward CASES construct. Let <A,[I],E> represent the array A after the assignment A[I]←E has been performed. Thus, we have

<a,[i],e>[i]=e and i≠j⊃<a,[i],e>[j]=a[j].

This fact can be written as a REPLACE rule

ARRAY: REPLACE <a,[i],e>[j] BY CASES i=j → e; i≠j → a[j] END;

This rule is equivalent to

ARRAY1: WHENEVER <a,[i],e>[j] FROM TRUE INFER CASES
i=j → <a,[i],e>[j]=e;
i≠j → <a,[i],e>[j]=a[j] END;

Interpret the "arrow" in the CASES clause as AND. Suppose we wish to prove

<A,[I],A[I]>[J]=A[J].

The rule splits and considers two cases

<A,[I],A[I]>[J]≠A[J] ∧ I=J ∧ <A,[I],A[I]>[J]=A[I]
<A,[I],A[I]>[J]≠A[J] ∧ I≠J ∧ <A,[I],A[I]>[J]=A[J]

Both cases simplify to *FALSE*, proving the theorem. In this example, the case split is required for proof.

Suppose, however, we were proving

<<B,[1],2>,[2],3>[1]=2.

If splits were done, four cases would be considered, three of which would be eliminated trivially. To avoid unnecessary splitting, and unnecessary delay of assertions of facts from forward rules, cases can be eliminated dynamically once a split has been propagated. As soon as only one case remains, its facts are asserted immediately. In our example, the rule ARRAY first applies with a=<B,[1],2>, i=2, e=3, j=1. The first case, with i=j or 2=1, is eliminated by test as soon as the rule applies, causing the other branch of the split to be propagated as fact. Thus the data base becomes,

<<B,[1],2>,[2],3>[1]≠2 ∧ 2≠1 ∧ <<B,[1],2>,[2],3>[1]=<B,[1],2>[1].

At this point, the rule applies again, with a=B, i=1, e=2, j=1. The second case is eliminated by test, and the fact <B,[1],2>[1]=2 is propagated causing an immediate contradiction.

Thus, forward CASES generally cause splitting only when a split is necessary or further splits are required to eliminate cases. Forward splits will not occur, however, within the proof of a subgoal for a backwards rule, though case elimination may cause facts to be asserted.

For efficiency reasons, only literals appearing before the arrow in a given case are used to eliminate other cases. Thus, the rule

CAS: FROM P INFER CASES Q1 → R1; Q2 → R2; Q3 → R3 END;

is equivalent to the set of rules

CAS1: FROM P ∧ ¬Q2 ∧ ¬Q3 INFER Q1 ∧ R1;
CAS2: FROM P ∧ ¬Q1 ∧ ¬Q3 INFER Q2 ∧ R2;
CAS3: FROM P ∧ ¬Q1 ∧ ¬Q2 INFER Q3 ∧ R3;

assuming it never splits. If any of the Qi or Ri are not fully instantiated when the split is propagated, further instantiations will occur only within each case, not across cases. The following two rules are equivalent:

CS1: FROM P(x) INFER CASES Q1(x,y); Q2(x,y) END;
CS2: FROM P(x) INFER CASES Q1(x,y); Q2(x,z) END;

4.2.12 Semantic matching

Suppose we had a rule

SM1: INFER P(x+1) FROM P(x) ∧ Q(x);

If we wanted this rule to apply in proving P(2+A) from P(A+1) and Q(1+A), the pattern matcher would need to have some knowledge of properties of addition. We call this type of matching Semantic Matching. The matcher used in the rulehandler makes use of properties of addition, multiplication, arithmetic relations, and equality. The matcher assumes that all variables appearing in sums and products are integer valued. This is conservative in the sense that no additional matches are obtained by the assumption, while many are eliminated.

Properties of addition and multiplication used are commutativity, associativity, identity, and in the case of addition, multiplication by constants. In the case of the relational operators, the integer assumption makes X≥0 and X+1>0 equivalent. In fact, the prover stores all inequalities of a given sign internally in the form E≥0 for some expression E. This means that the pattern F(x)≥y will match A+B<F(C)+G(D), binding x to C and y to A+B-G(D)+1. Note that only a negated inequality pattern will match a negated inequality in the data base, however.

Equality matching makes use of the symmetric and substitutive properties of equality. Thus, the rule

EQ1: INFER x=y FROM P(x,y);

will prove P(B,A) ⊃ A=B.

Often, patterns will not match as soon as the target is found because facts asserted later in proof are required for the match. For example, in proving P(A) ∧ A=F(B) ⊃ Q(B) with the rule

t1: FROM P(F(x)) INFER Q(x);

the rule applies only after A=F(B) has been asserted to the data base. For efficiency reasons, this sort of "waiting" does NOT take place with semantic patterns. Thus, P(A*B) ∧ A=F(C) ⊃ Q(B,C) will not be proved by

Q2: FROM P(x*F(y)) INFER Q(x,y);

while A=F(C) ∧ P(A*B) ⊃ Q(B,C) will. This limitation is not a serious one in practice, and may be circumvented by using a WHENEVER clause, as in

Q3: WHENEVER F(y) FROM x*F(y) INFER Q(x,y);

4.2.13 Subspace matching

When matching a pattern like x+y against a sum, it is possible that many distinct matches will result. For this reason, certain sum matches produce "subspace" specifications as their result. For example, matching x+y against A+B+C produces the specification x+y=-(A+B+C), which represents a linear equation with variables x and y. When x or y appear in further patterns, they will be considered to be unbound, except subject to the constraint of this equation. Multiple constraints are merged using Gaussian Elimination over the integers. Thus P(x+y, x-y) will match P(A,A-4*B), binding x to A-2*B and y to 2*B. P(x+y, x-y) and P(A,A-3*B) will not match, however, because x and y are considered to be integer variables.

Subspace matching is a powerful facility, but it is not desirable in certain instances. Consider the rule

DIST: REPLACE (a+b)*c BY a*c + b*c;

Since a and b will be part of a subspace specification, the BY clause will not be instantiated, severely limiting applicability of the rule. For this reason, a facility has been provided which allows extra specific instances to be generated by the matcher in addition to the subspace specifications. This facility is controllable by a switch (called SUMMATCH), since for efficiency reasons, it may not always be desirable.

In some cases, it may be necessary to eliminate the subspace match entirely. If we were simplifying

P(A) ∧ P(3) ∧ P(B) ∧ P(C) ⊃ P(C+B)

with the rule

SUMP: INFER P(x+y) FROM P(x) ∧ P(y);

many unnecessary subgoals would be generated. When the rule matches, it generates a subspace specification for x+y=(C+B) in its virtual instance. FIND is used to locate instances of P(x), since x is not fully specified. The first instance is P(A), resulting in the binding of x to A and y to C+B-A by solving through. By this process, many useless goals will be generated. If we could somehow guarantee that P(x) would be instantiated, we would not have this problem. One way to do this is to invent a new predicate which does not appear in the theorems to be proved. Suppose we replaced SUMP by

SUMP2: INFER P(x+y) FROM INST(x) ∧ P(x) ∧ P(y);
SUMAUX: INFER INST(x);

Since INST does not appear in the data base, it can only be proved with rules. But rules only "fire" on instantiated literals, so FIND will always fail on INST, eliminating the subspace matches. Thus, only the specific instances (provided as a result of setting the switch mentioned above) will be considered. This combination of rules guarantees that P(x) (and consequently P(y)) in SUMP2 will be instantiated.

4.2.14 Efficiency considerations

The user is reminded that the theorem prover is limited in its capacity. Rules may be thought of as a device for programming the theorem prover: it is easy to write inefficient programs -- harder to write efficient ones. Like programs, inefficient rulefiles cause the prover to use excessive time or space, running until either the patience of the programmer or core storage is exhausted. This sort of inefficiency can be prevented in many cases by merely considering efficiency as well as logical elegance when writing rulefiles. Remember, however, that there are many concepts that are difficult to code effectively as rules.

Beware of excess searching caused by badly ordered backward rules. When writing rulefiles, consider how to order the rules so search will be efficient. Simply reordering rules and literals within rules can lead to dramatic decreases in proof times.

Beware of forward rules asserting multitudes of useless facts and causing unnecessary splits. Strengthen FROM clauses to restrict application.

Beware of rules that create numerous virtual instances. For example,

LOSS: WHENEVER F(x), F(y) FROM F(x)=F(y) INFER P(x,y);

will create $n\uparrow 2$ instances of the rule if there are n instances of F(-) in the data base. While most of the virtual instances may not fire, their presence in the data base will increase the space required for proof.

Plan carefully whether to use forward or backward rules, or both, to express a particular concept. Forward rules are most effective for "complete" domains, where all relevant facts can be propagated immediately. Examples of such domains are simple data structures, type properties of program data objects, and some simple arithmetic facts. Backward rules are best suited to larger "incomplete" concepts, where forward inference would produce too many facts or could not generate all relevant facts. Ordered and Permutation are examples of such concepts.

Adjust the depth bounds to conservative values before attempting a large proof. Some domains with very broad search spaces need shallow bounds, while other domains which require narrow, deep searching need to have the bounds set accordingly. Rules which require broad deep searches will be inefficient; it may be advisable to rethink their structure.

In general, the best advice is to understand what a set of rules means from both the heuristic and logical viewpoints. Syntactically translating logical statements into rules without regard to efficiency can lead to prolonged and wasteful searches.

4.2.15 A note on multiplication

The built in Presburger arithmetic package (which is independent of the rulehandler and semantic matcher) includes a facility for recognizing multiplication by constants. However, this facility is equivalent to the set of rules:

```
REPLACE -1*x BY -x
REPLACE 0*x BY 0
REPLACE 1*x BY x
REPLACE 2*x BY x+x
```

and so on. This means that without rules, the formula $P(x*y) \wedge x=0 \supset P(0)$ will not simplify, while the formula $x=0 \wedge P(x*y) \supset P(0)$ will simplify. This unfortunate weakness can often be circumvented partially by adding rules of the sort:

```
REPLACE x*y WHERE x=1 BY y
REPLACE x*y WHERE x=2 BY x*y
```

and so on, where necessary.

REFERENCES

[1] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation, and Compiling, Vol. I, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

[2] W.W. Bledsoe, Splitting and reduction heuristics in automatic theorem proving, Artificial Intelligence, Vol. 2, 1971, 55-77.

[3] R. Cartwright, and D.C. Oppen, Unrestricted procedure calls in Hoare's logic, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York, 1978.

[4] S.A. Cook, Axiomatic and interpretive semantics for an Algol fragment, Technical Report 79, University of Toronto, 1975.

[5] R.L. Drysdale,and H.J. Larsen, A standard basis for automatic verification of sorting algorithms, forthcoming AI Memo, Stanford Artificial Intelligence Laboratory, Stanford University.

[6] D.A. Fisher, Copying cyclic list structures in linear time using bounded workspace, CACM, Vol. 18, 5, May 1975, 251-252.

[7] R.W. Floyd, Assigning meanings to programs, Proc. Symp. Appl. Math. Amer. Math. Soc., Vol. 19, 1967, 19-32.

[8] S.M. German, Automating proofs of the absence of common runtime errors, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York, 1978, 105-118.

[9] S.M. German, D.C. Luckham, and D.C. Oppen, Proving the absence of common runtime errors, forthcoming AI Memo, Stanford Artificial Intelligence Laboratory, Stanford University.

[10] F.W. v.Henke, and D.C. Luckham, A methodology for verifying programs, Proceedings of the International Conference on Reliable Software, Los Angeles, California, April 20-24, 1975, 156-164.

[11] C.A.R. Hoare, An axiomatic basis for computer programming, CACM, Vol. 12, 10, Oct. 1969, 576-580, 583.

[12] C.A.R. Hoare, Proof of a program: FIND, CACM, Vol. 14, 1, Jan. 1971, 39-45.

[13] C.A.R. Hoare, and N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Informatica, Vol. 2, 1973, 335-355.

[14] S. Igarashi, R.L. London, and D.C. Luckham, Automatic program verification I: Logical basis and its implementation, Acta Informatica, Vol. 4, 1975, 145-182.

[15] K. Jensen, and N. Wirth, Pascal User Manual and Report, second ed., Springer-Verlag, New York, 1975.

[16] R.A. Karp, and D.C. Luckham, Verification of fairness in an implementation of monitors, Proceedings International Conference on Software Engineering, San Francisco, Oct. 1976, 40-46.

[17] J. King, and R.W. Floyd, Interpretation oriented theorem prover over integers, Second ACM Symposium on Theory of Comp., Massachusetts, 1970.

[18] D.E. Knuth, The art of computer programming, Vol. III - Sorting and Searching, Addison-Wesley Publishing Company, Reading, Mass. 1973.

[19] D.C. Luckham, and N. Suzuki, Automatic program verification IV: Proof of termination within a weak logic of programs, AI Memo AIM-269, Stanford Artificial Intelligence Laboratory, Stanford University, Oct. 1975; also, Acta Informatica, 8, 1977, 21-36.

[20] D.C. Luckham, and N. Suzuki, Automatic program verification V: Verification-oriented proof rules for arrays, records, and pointers, AI Memo AIM-278, Stanford Artificial Intelligence Laboratory, Stanford University, March 1976; revised: "Verification of array, record, and pointer operations in Pascal", Dec. 1978.

[21] Z. Manna, Mathematical Theory of Computation, McGraw-Hill Book Company, New York, N.Y., 1974.

[22] C.G. Nelson, and D.C. Oppen, Fast decision procedures based on congruence closure, AI Memo AIM-309, Stanford Artificial Intelligence Laboratory, Stanford University, Feb. 1978; also, Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science, 1977.

[23] C.G. Nelson, and D.C. Oppen, Simplification by cooperating decision procedures, AI Memo AIM-311, Stanford Artificial Intelligence Project, Stanford University, April 1978; also, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York, 1978.

[24] D.C. Oppen, Reasoning about recursively defined data structures, Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, ACM, New York, 1978.

[25] W. Polak, Verification of the in-situ permutation program, forthcoming IEEE Software Engineering, July 1979.

[26] N. Suzuki, Verifying programs by algebraic and logical reduction, Proceedings of Int'l. Conf. on Reliable Software, IEEE, Oct. 1975, 473-481.

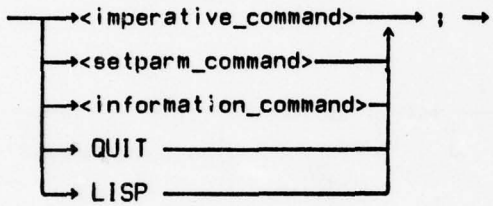
[27] N. Suzuki, Automatic verification of programs with complex data structures, Ph.D Thesis, Computer Sci. Dept., Stanford University, 1976.

Alternate notation

assignment	← or :=	less or equal	≤ or <=
greater or equal	≥ or =>	not equal	≠ or <>
implication sign	→ or ->	and	∧ or &
or	∨ or ! or |	reference class extension	∪ or &&
negation	¬ or ~	reference class selection	⊂ or [\ , ⊃ or \]
history sequence concatenation	• or !! or ||		

A.1 Command syntax

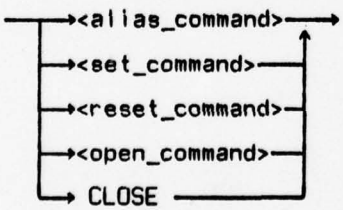
<command>



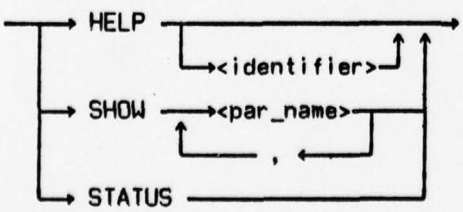
<imperative_command>



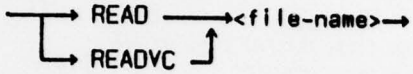
<setparm_command>



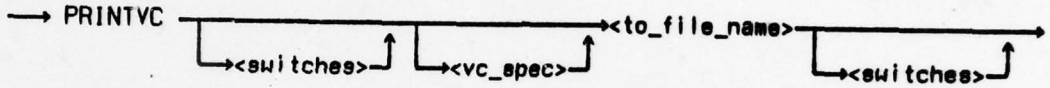
<information_command>



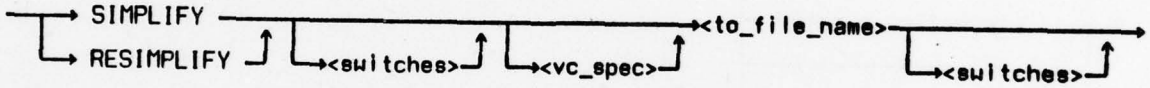
<read_command>



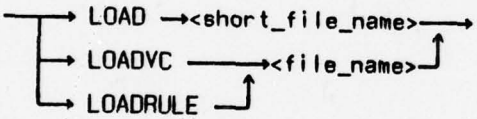
<print_command>



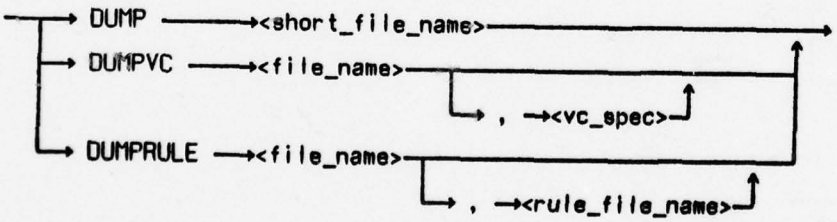
<simp_command>



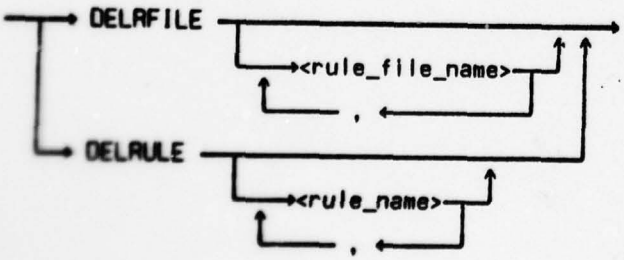
<load_command>



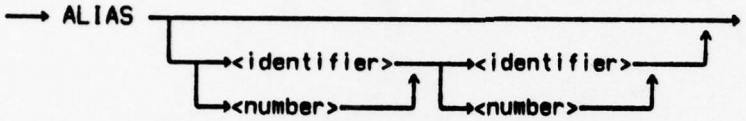
<dump_command>



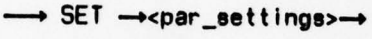
<delete_command>



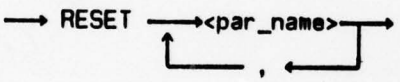
<alias_command>



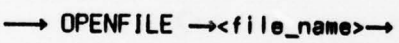
<set_command>



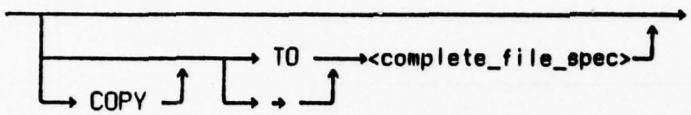
<reset_command>



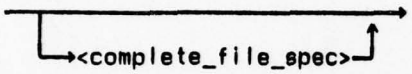
<open_command>



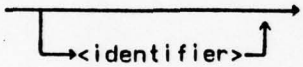
<to_file_name>



<file_name>

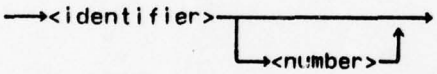


<short_file_name>



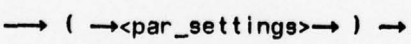
<complete_file_spec> is the standard monitor syntax for a file name.

<vc_spec>

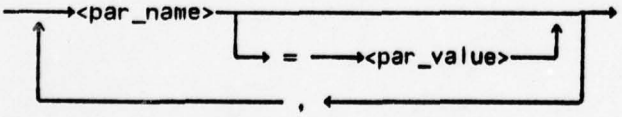


<rule_name>, <rule_file_name>

<switches>



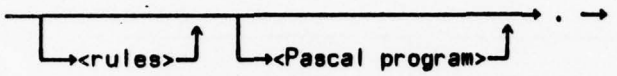
<par_settings>



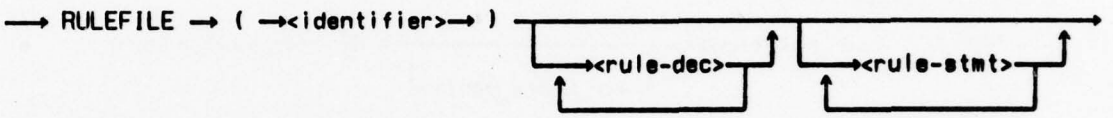
<par_name>, <par_value> refer to the list of System Parameters given in Part II, Section 2.2.

A.2.1 Outer level of input

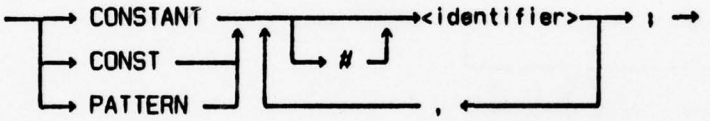
<source input>



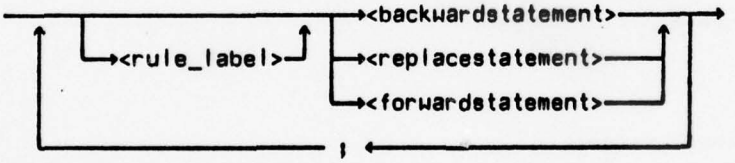
<rules>



<rule-dec>

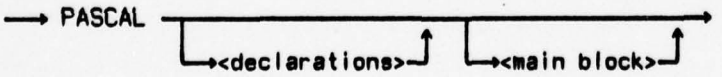


<rule-stmt>



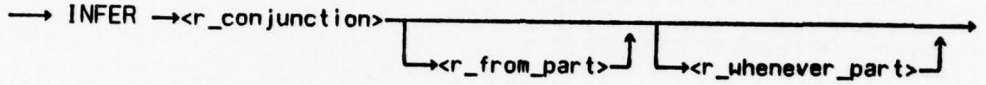
<rule_label>

<Pascal program>

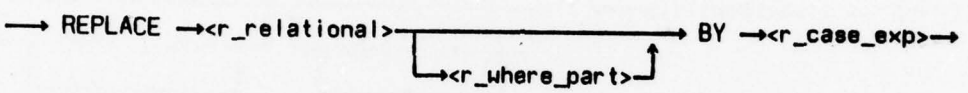


A.2.2 Statements that appear in rulefiles

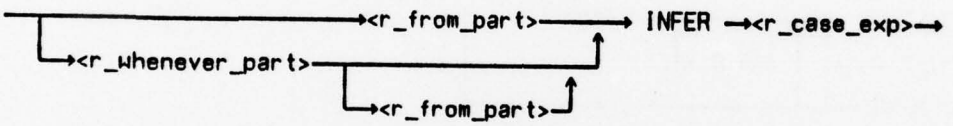
<backwardstatement>



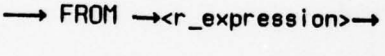
<replacestatement>



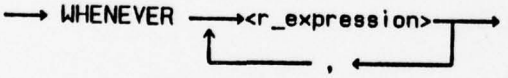
<forwardstatement>



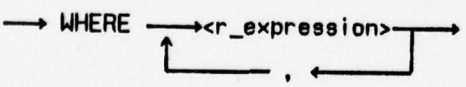
<r_from_part>



<r_whenever_part>

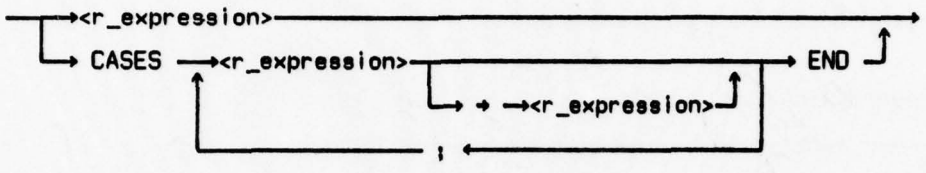


<r_where_part>

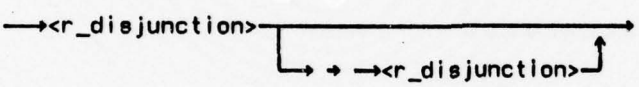


A.2.3 Expressions that appear in rulefiles

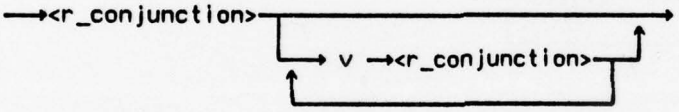
<r_case_exp>



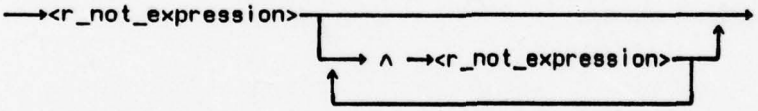
<r_expression>



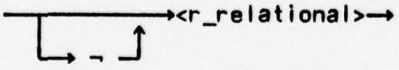
<r_disjunction>



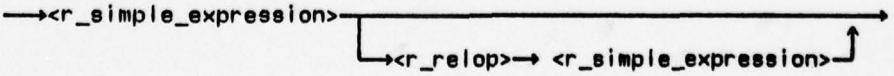
<r_conjunction>



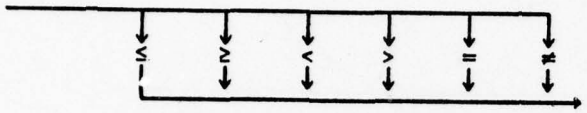
<r_not_expression>



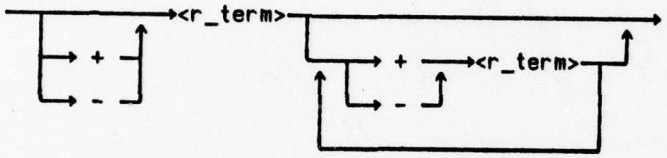
<r_relational>



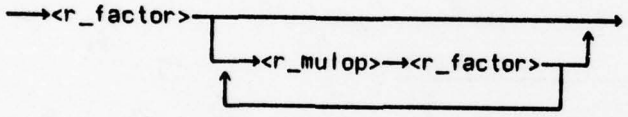
<r_relop>



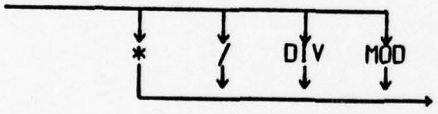
<r_simple_expression>



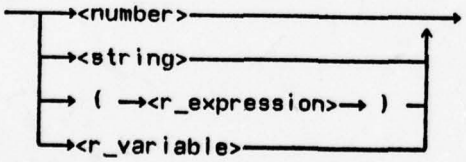
<r_term>



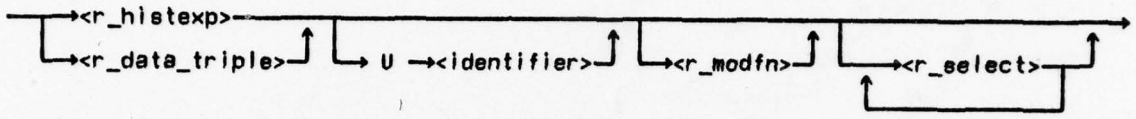
<r_mulop>



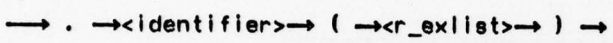
<r_factor>



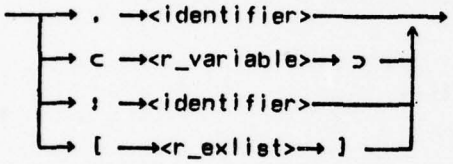
<r_variable>



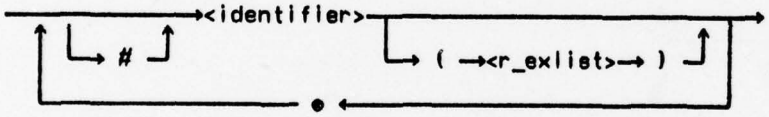
<r_modfn>



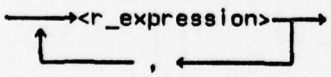
<r_select>



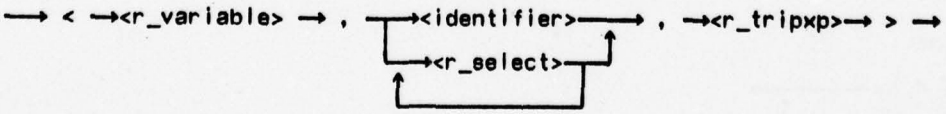
<r_histexp>



<r_exlist>



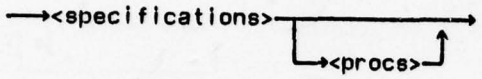
<r_data_triple>



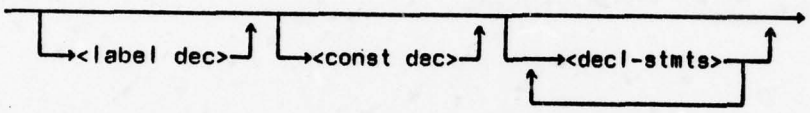
<r_tripxp> is the same as <r_expression> except that at the level of r_relop, the relational operator ">" is omitted. This has the effect that expressions containing this operator must be enclosed in parentheses when appearing in the final portion of a data triple. It is required to eliminate ambiguities caused by using > to terminate a data triple.

A.2.4 Outer structure of Pascal programs

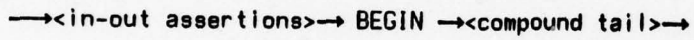
<declarations>



<specifications>

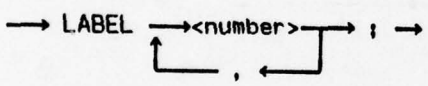


<main block>

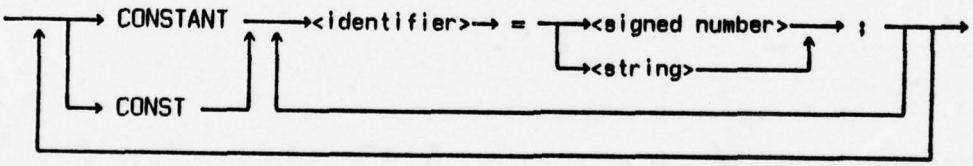


A.2.5 Nonexecutable statements

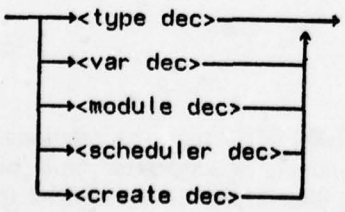
<label dec>



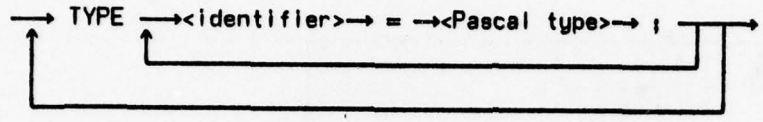
<const dec>



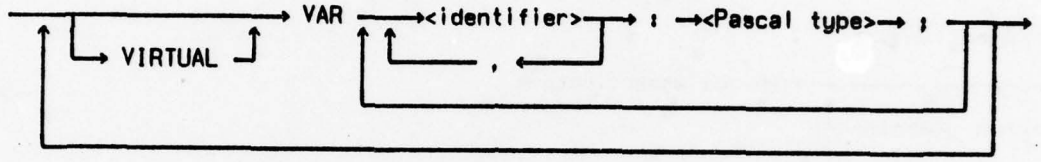
<decl-stmts>



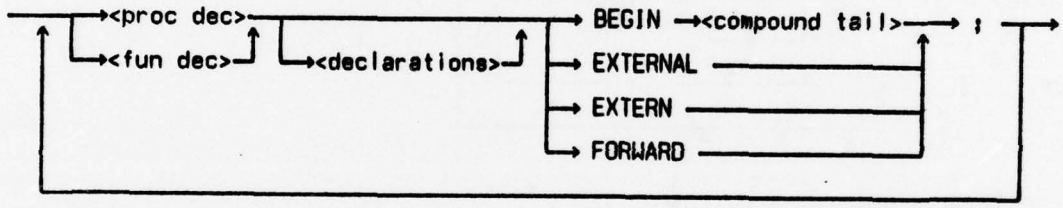
<type dec>



<var dec>

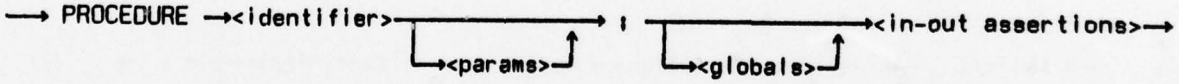


<procs>

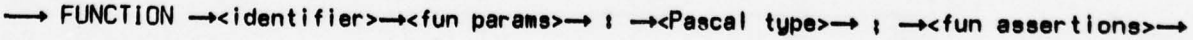


A.2.6 Procedure declarations and associated assertions

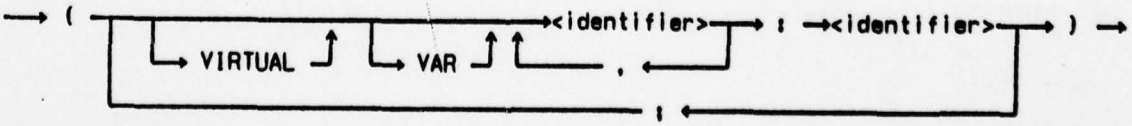
<proc dec>



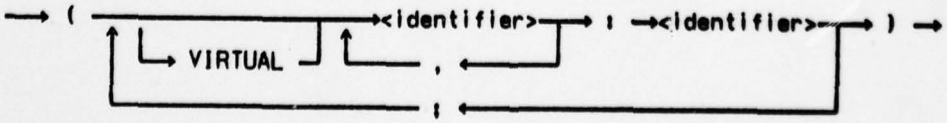
<fun dec>



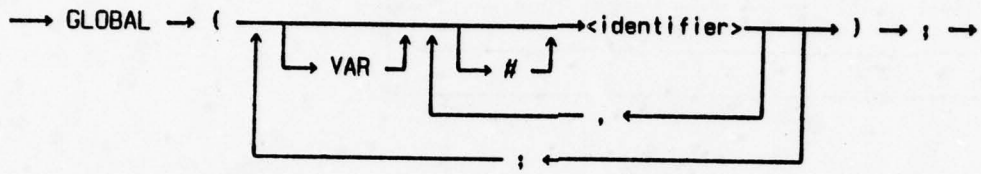
<params>



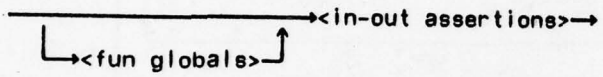
<fun params>



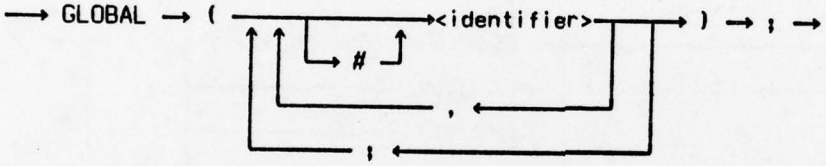
<globals>



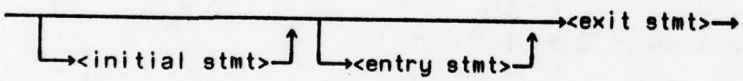
<fun assertions>



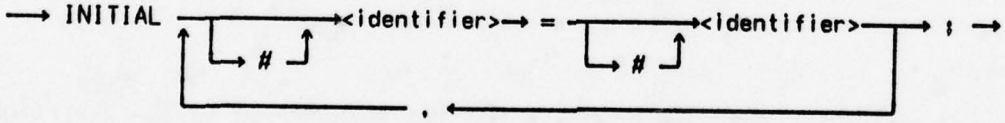
<fun globals>



<in-out assertions>



<initial stmt>



<entry stmt>

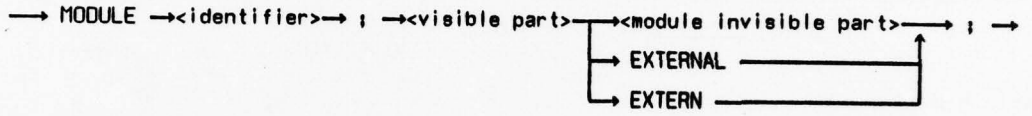
⟶ ENTRY ⟶<a_expression>⟶ ; ⟶

<exit stmt>

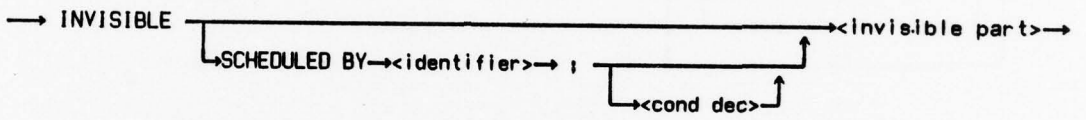
⟶ EXIT ⟶<a_expression>⟶ ; ⟶

A.2.7 Module and Scheduler declaration

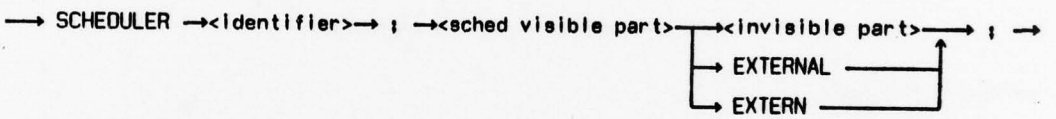
<module dec>



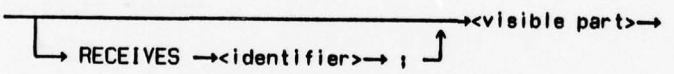
<module invisible part>



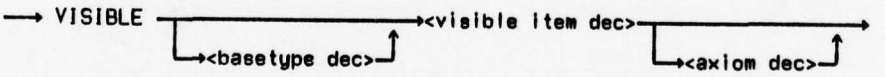
<scheduler dec>



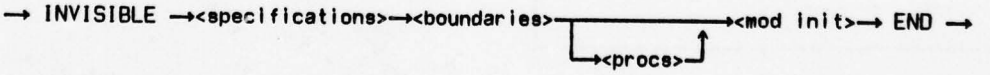
<sched visible part>



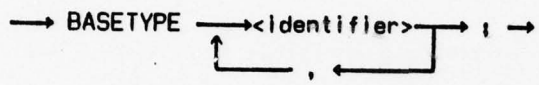
<visible part>



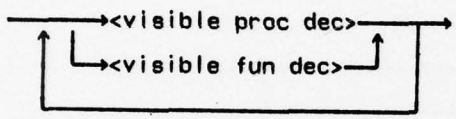
<invisible part>



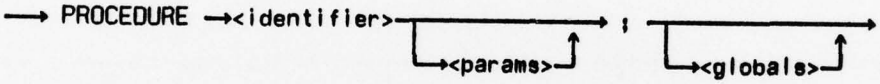
<basetype dec>



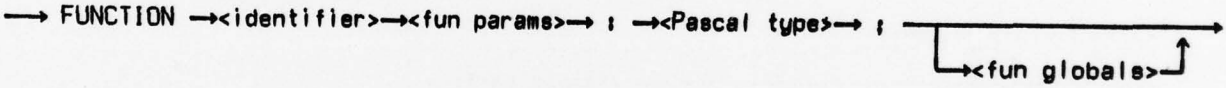
<visible item dec>



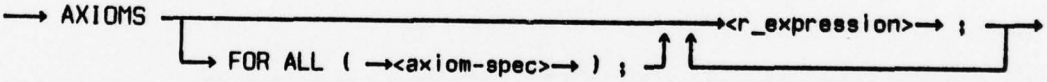
<visible proc dec>



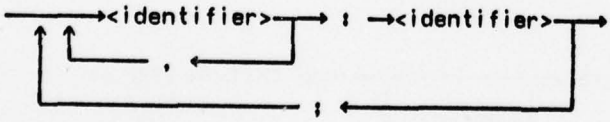
<visible fun dec>



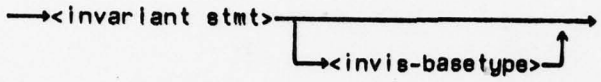
<axiom dec>



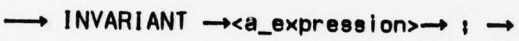
<axiom-spec>



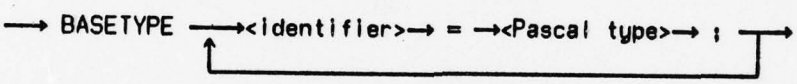
<boundaries>



<invariant stmt>



<invis-basetype>

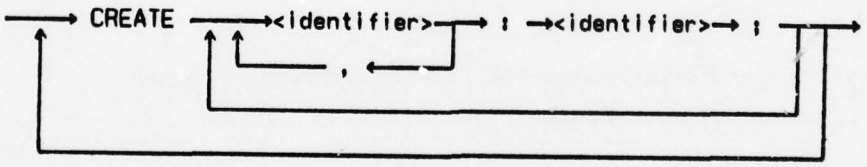


<mod init>

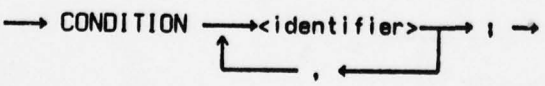


A.2.8 Module and condition variable instantiation

<create dec>

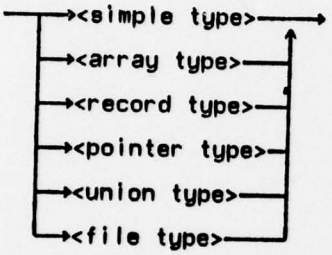


<cond dec>

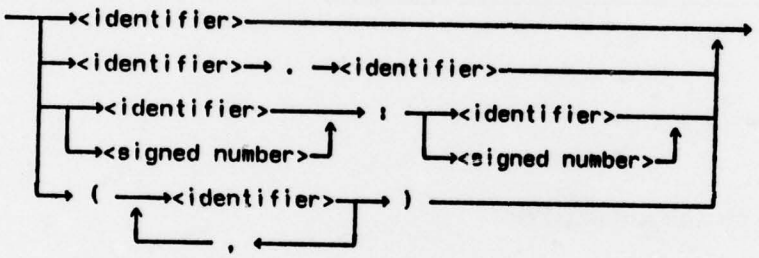


A.2.9 Pascal type declarations

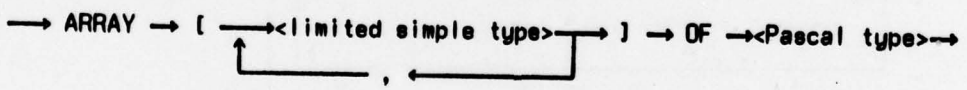
<Pascal type>



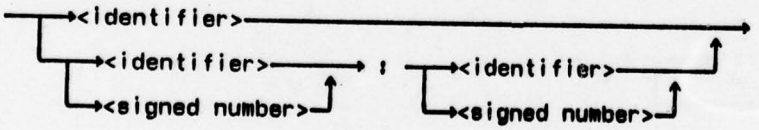
<simple type>



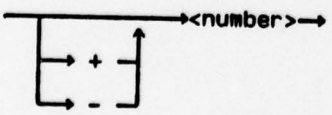
<array type>



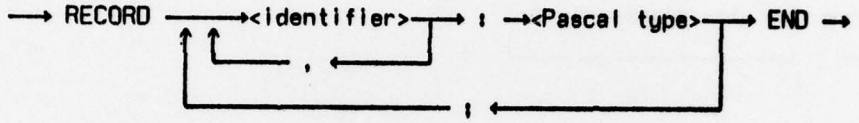
<limited simple type>



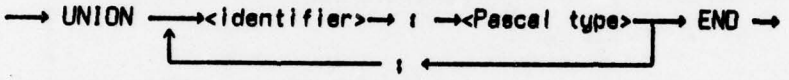
<signed number>



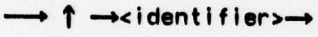
<record type>



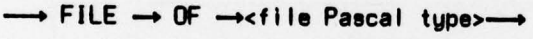
<union type>



<pointer type>



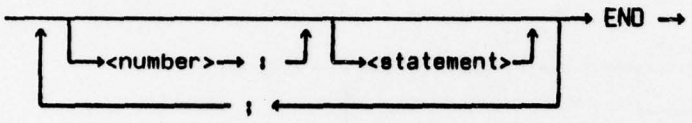
<file type>



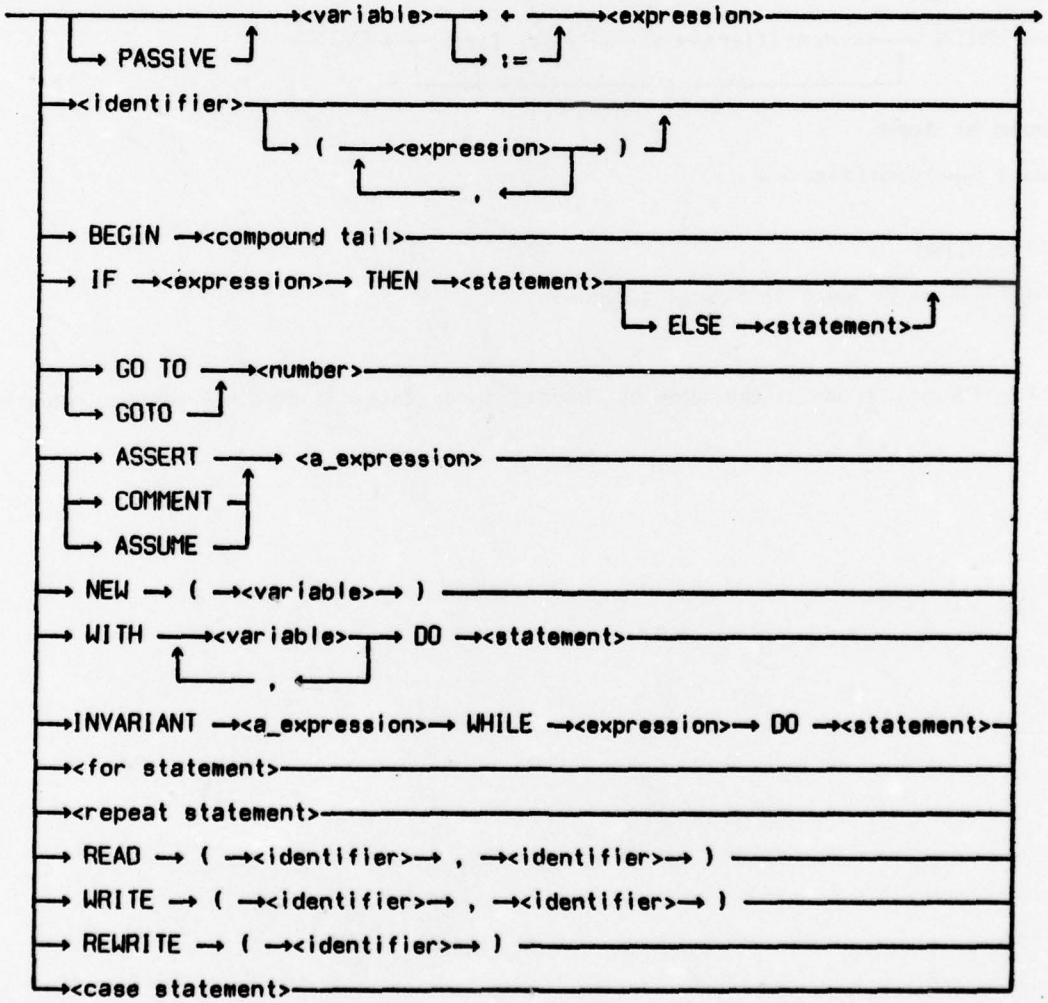
<file Pascal type> is the same as <Pascal type> except it does not contain <pointer type>.

A.2.10 Executable statements

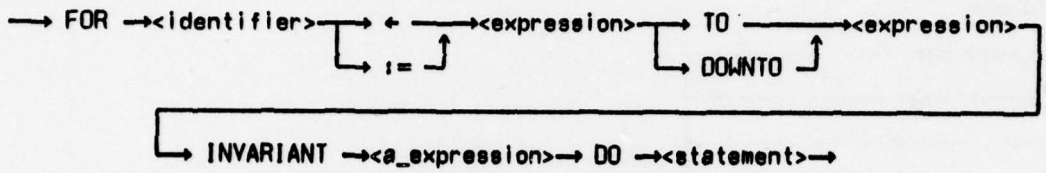
<compound tail>



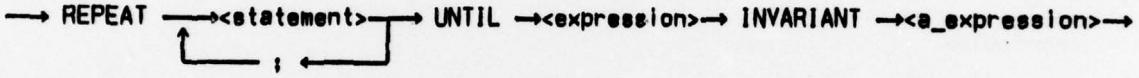
<statement>



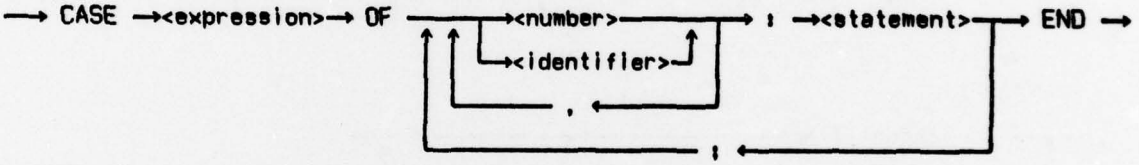
<for statement>



<repeat statement>

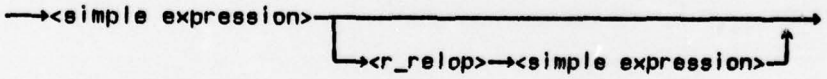


<case statement>

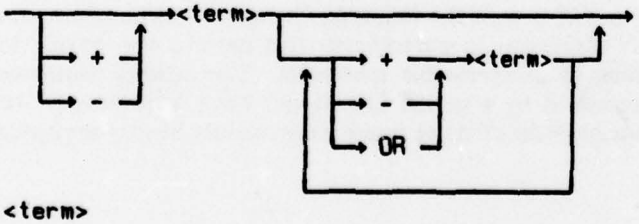


A.2.11 Expressions in Pascal programs

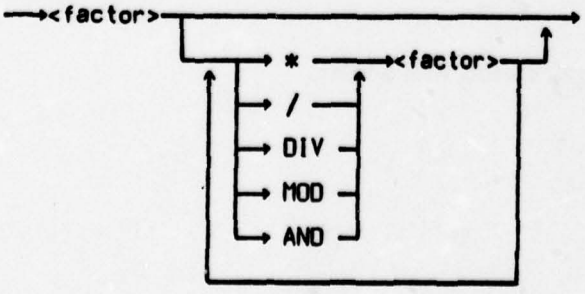
<expression>



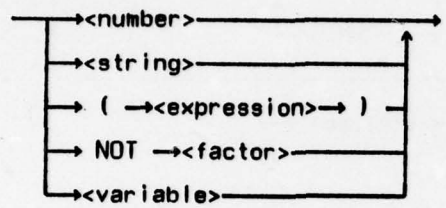
<simple expression>



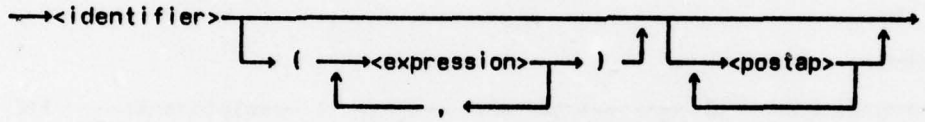
<term>



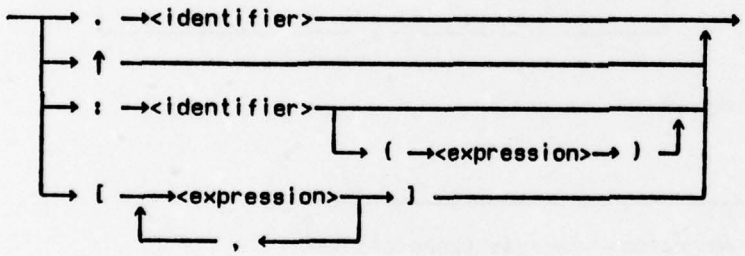
<factor>



<variable>



<postap>



<a_expression> is the same as <r_expression> with the following changes--A union selection :<identifier> may be followed by an expression in parentheses; this permits the parser to automatically build the union construction, as in executable statements. The history sequence operator • is prohibited; record fields indicated by a period (.) may not have a parameter list following the fieldname. These restrictions have the effect of prohibiting module history sequence statements.

<number> is an unsigned constant.

<string> is a character string.

<identifier> is a sequence of letters and digits, starting with a letter.

B.1 General

The parser makes a pass over the source code you have provided for correct syntax. If this results in no error, the message "SYNTAX SCAN COMPLETE" is given. If an error occurs, the parser will tell you what it was scanning, what would have been an acceptable next token, and what some previous tokens were.

This initial syntax scan merely verifies that the format of what is seen is correct; it makes no checks on the actual content. If this syntax scan is satisfactory, a second phase is entered where content checks are made. What follows is a list of errors that can occur during this second or semantic phase. If this second phase is completed successfully, then whatever action the parser was trying for you is then done. Note that when parsing Pascal code, verification conditions for procedures and functions which were completely parsed prior to a semantic error will be present and can be still worked on with the simplifier.

The following listing is in alphabetic order. The notation "vcg" following a message indicates that the source of the error is the verification condition generator rather than the parser. This should not normally be of concern to a user.

B.2 Semantic errors

ACTUAL PARAMETER TYPE DOES NOT MATCH FORMAL DECLARATION

The parser checks procedure and function calls to ensure that the type of each parameter matches the declaration of that procedure or function. One of yours didn't make it. Information printed out may include the type expected or the name of the formal parameter in the declaration.

ARGUMENT LIST EXPECTED

A function name appeared in an expression and it was not followed by an argument list enclosed in parentheses.

BAD PUT ENTRY--VERIFIER ERROR

An internal check in the parser symbol table entry code has discovered something that shouldn't be there. If this was a program product of some manufacturer, you'd be instructed at this point to send in a trouble report. As it is, the choices are less appealing! In any case, it would be bad to trust anything produced by the parser after getting this error.

BASE TYPE FOR POINTER NOT DEFINED

In an assertion within a Pascal program, you used the notation "<identifier>↑". To correctly translate this into an assertion the system understands, the parser has to be able to figure out what reference class the <identifier> belongs to. It does this by looking up the entity in its symbol table, and in this case it couldn't find it. If you want to include this as part of the assertion, you will have to provide the reference class. Instead of this syntax, use *<base type> ⊂ <identifier> ⊃ (no blanks between * and <base type>).

BASE TYPE FOR REFERENCE CLASS DOES NOT MATCH WHAT WAS EXPECTED

In an assertion within a Pascal program, you used the notation "*<identifier1> ⊂ <identifier2> ⊃" (or some qualified form equivalent to this). Either <identifier2> was not of pointer type, or if it was of pointer type, its base type was not the same as <identifier1>.

BOOLEAN EXPRESSION EXPECTED

An expression of boolean type was expected, such as in a WHILE test or an IF test.

BOTH SIDES OF ASSIGNMENT MUST BE COMPATIBLE TYPES

For an assignment statement to be correct, the types of the entity being stored into and the type of the expression being stored must be compatible. Thus, they must both be numbers, or one must be a subset of the other, or they must be the same type. You had an assignment statement where this was not the case.

BRANCHING INTO COMPOUND STATEMENTS PROHIBITED

You may not branch into a WHILE, REPEAT, FOR, or WITH body using the GOTO statement. If you need unlimited branching, you will have to create your control structure entirely with GOTO not using any of these iteration statements.

CASE NAME TYPE MUST MATCH CASE EXPRESSION

At the head of a CASE statement is an expression of a certain type. Each of the cases following must be identified with a constant of the same type.

CHAR TYPE MAY ONLY HAVE ONE CHARACTER STRINGS

An entity of type CHAR may be a string at most one character long. Longer strings will be allowed eventually.

CLASS NAME INCORRECTLY QUALIFIED OR USED

A class name must be followed by a period and another identifier when invoking a procedure or function from the class externally. Alternatively, you tried to assign to a class procedure name or function.

CONSTANT DUPLICATED IN THIS TYPE DEFINITION

A union type consists of ids followed by types; each of these ids must be distinct within a given type definition. You duplicated one of the ids.

CONSTANT MAY NOT BE QUALIFIED

You have an identifier which was given a value in a CONST or CONSTANT statement. These identifiers have the value you gave substituted by the parser, thus they are really parse time abbreviations. In particular, they are always scalars and can't be subscripted, or have record fields, etc. following them.

CONSTANT MAY NOT BE STORED INTO

You have an identifier which was given a value in a CONST or CONSTANT statement. These identifiers have the value you gave substituted by the parser; thus they are really parse time abbreviations. Therefore, you can't store into them -- put them on the left hand side of an assignment statement except as part of a subscript or something like that.

CONSTANT OF A KNOWN ENUMERATED TYPE EXPECTED

Each union type consists of a list of id-type pairs. Each id must be a constant of the same enumerated type. You have given an id which is not a constant of an enumerated type.

CONSTANT TYPE DIFFERS FROM PREVIOUS CONSTANTS

Each union type consists of a list of id-type pairs. All the ids must be constants of the same enumerated type. You have given an id of a different type than previously encountered in this declaration.

DUPLICATE LABEL IN CASE STATEMENT

The same label appears twice in a case statement. Each case must appear at most once.

EMPTY CASE STATEMENT (vcg)

This message should not be printed under any circumstances. If it does occur, it indicates that the parser has produced a case statement with no branches.

ERROR IN ASSIGNMENT STATEMENT (vcg)

This message should not be printed under any circumstances. If it does occur, it indicates that the parser has produced an illegal assignment statement.

ERROR IN C_D_U - CASE 1

Caused by forgetting to set _ to T.

ERROR IN C_D_U - CASE 2

Caused by forgetting to set _ to NIL.

ERROR IN C_D_U - THIRD TYPE

Now you've really done it. You were warned NOT to use the CONCURRENT Dynamic Underbar feature UNLESS you talked to me first. Now that you are having trouble, don't expect me always to solve YOUR problem.

This message MIGHT also be caused by incompleteness in the W matcher, so be sure to send a complete minimal protocol to BUG-VERIFY % STANFORD, zip code 94305, and allow at least nine months for delivery..

EXIT ASSERTION OMITTED FROM PROCEDURE OR FUNCTION

An EXIT assertion is required by the system. The absence of one is usually detected by the syntax scan. But when the word PROCEDURE or FUNCTION followed by just a name is found, the syntax scan must permit it since it could be the body of a block declared forward. If it isn't, this error is given.

FILES CANNOT APPEAR IN ASSIGNMENT STATEMENTS

You tried to assign to an identifier of type FILE. Files may appear only in assertions, READ statements, and WRITE statements (in addition to being declared).

FILES OF ENTITIES OF POINTER TYPE ARE PROHIBITED

The base type of a file may not be of type POINTER.

FOR CONTROL VARIABLE MAY NOT BE REDEFINED INSIDE FOR STMT

Pascal prohibits redefining the FOR statement control variable within its loop. Note that this error can occur when the control variable is passed to a procedure which may change it -- i.e., as a VAR parameter or when it is declared as GLOBAL within the called procedure. It can also occur in more obvious ways.

FUNCTION NAME MAY NOT BE USED AS VARIABLE (vcg)

You have a function or predicate name appearing in an assertion or code which is also declared as the name of a variable. This is not permitted.

FUNCTIONS MAY NOT HAVE SIDE EFFECTS--STRICT ENFORCEMENT

In order to permit only functions without side effects, the parser is extremely rigid in disallowing things. In particular: function bodies may not contain global statements, IO statements, or NEW statements. In addition, functions may not have VAR parameters. This rather severly limits functions! You may have to make your function into a procedure which returns its value as a VAR parameter. Sorry!

GENSYM AND YOU AGREE--SORRY!--RENAME YOUR VARIABLE

When the parser called the LISP function GENSYM to invent a name for some reason or another, the name returned was already in your program, declared as one of your entities in this block. You must change the name of the entity of that type. This message will usually be given in addition to an IDENTIFIER DECLARED MULTIPLY message.

GLOBALS FROM OUTSIDE THE MODULE MUST APPEAR IN VISIBLE GLOBAL STMT

Module visible procedures may have two global statements: one, appearing with the visible declaration, describes the entities global to the module that the procedure might change. The second, attached to the invisible declaration of the procedure, details the module variables changed by this procedure.

IDENTIFIER DECLARED MULTIPLY IN ONE BLOCK

This particular identifier is already the name of something in this block. Change one or the other.

ID IN POINTER OF INCORRECT TYPE

When defining a pointer type, the pointer base type must be another type identifier. Since the base type for a pointer type may appear before it is defined, this error may not appear until after processing all TYPE statements for a particular function or procedure.

ID NOT DECLARED OR NOT A VARIABLE

In processing an expression, the parser found an identifier that was not in the symbol table; or if it was, it was not declared as a VAR but rather was of some other kind. This error can occur, for example, if a virtual variable appears in executable code (other than documentation or a PASSIVE statement).

ID NOT DECLARED AS VISIBLE BASETYPE NAME

In the BASETYPE specification within the invisible part of a module, you tried to declare the specifications of an identifier that was not declared as the name of a basetype in the visible specifications.

ILLEGAL ENTRY ASSERTION FOR FUNCTION (vcg)

The ENTRY assertion for a function may not contain the function name.

ILLEGAL PROCEDURE CALL (vcg)

The procedure call rule requires that each of the VAR parameters and GLOBAL variables in a particular procedure call refer to a distinct variable.

IMPROPER SUBRANGE DEFINITION

Subranges may be declared as explicit types or as subscripts for arrays. They are usually two values, in which case the lower value of the subrange must really appear before the upper value in the defintion of the base type. In particular, for subranges of integers, the first integer must be smaller than the second. Also the types of the two entities in the subrange must be compatible with each other.

INTERNAL ERROR IN VCG OR RUNCHECK (vcg)

This message is produced only in the special runtime error checking version of the verifier. It indicates a system error in the verifier.

INVALID ARGUMENT TYPE TO ARITHMETIC OR LOGICAL OPERATOR

The parser checks that each arithmetic or logical operator only receives sub-expressions of proper type; thus * expects only to find two numbers, NOT a boolean, etc.

INVALID CONSTRUCTOR OR SELECTOR FOR UNION TYPE

Union type construction must have three elements: the type to be constructed, the tag to be associated with it, and the value to be associated with it, in that order. The type of the value must be consistent with the tag, and the value must be present. Therefore, there must be an expression enclosed in parenthesis of the appropriate type, and there must be a tag of the appropriate type. Union selection, however, merely consists of a union variable followed by selection of a union field. No expression may follow.

INVALID SUBRANGE ITEM

Subranges may be declared as explicit types or as subscripts for arrays. In the latter case only, a VAR is permitted. In both cases, a number, an abbreviation for a number (identifier defined in a CONST or CONSTANT statement), or a constant of an enumerated type may be used. None of these types of entities were found in your definition.

INVALID TYPE FOR CASE STATEMENT EXPRESSION

The expression following the keyword CASE must be of scalar type. Further, it may not be of type REAL or a subset of type REAL.

INVALID TYPE FOR CONTEXT WHERE USED

An attempt was made to dereference (↑) an entity not of type pointer, or subscript an entity not of type array. Alternatively, in a FOR statement, the index variable and both expressions must be compatible with a numeric type. Finally, too many subscripts were present for a particular var (i.e., there were two subscripts to an array which only had one dimension, or one subscript to a var that was not an array).

KNOWN TYPE NAME EXPECTED

When you define a type in terms of another type, the second type must already be known to the parser (exception: pointer base types). Also, the base type for a reference class appearing in a GLOBAL statement must be known to the parser. Finally, the type of a parameter to a function or procedure must be known to the parser before seeing the function or procedure declaration.

LABEL APPEARS IN PROGRAM BUT IS NOT DECLARED

You use a label in your program unit but do not declare it entering the program unit (with the LABEL declaration). Labels must be local to the procedure in which they appear, and must be declared there.

LABEL DECLARED AND REFERENCED BUT NOT PRESENT

Somewhere in your procedure or function, you have stated GOTO n, but after completing parsing your procedure or function, the label n was not found on any statement within that procedure or function. Note that if n is within the body of a nested procedure or function, it is not regarded as being within the body of the outer procedure or function.

LABEL MUST BE POSITIVE INTEGER

A label must be a positive integer; it cannot be zero.

LABEL SPECIFIED MULTIPLY IN ONE PROGRAM UNIT

The same label appears on two or more statements in one procedure or function.

MISSING ASSERTION ON PATH THROUGH LABEL (vcg)

The program contains a closed path formed by a GOTO, but there is no assertion anywhere in the path.

MISSING ITEM IN LOADSYMBOLS INVOCATION

The LOADSYMBOLS command contains two parameters. The first is the name of the procedure whose symbol table environment is being recreated; the second is the name of the file containing the symbol table code. One of these was missing.

MULTIPLY DEFINED IDENTIFER IN INVALID CONTEXT

Contrary to the usual scope rules, once an entity is defined to be a TYPE, MODULE, or SCHEDULER identifier, it may not be redefined as a TYPE, MODULE, or SCHEDULER identifier in a less global scope. Give the new type, module, or scheduler another name.

NAME OF MODULE EXPECTED

In a CREATE statement, the identifier following the colon must be the name of an entity previously defined as a MODULE.

NEW STATEMENT MUST HAVE POINTER ARGUMENT

A NEW statement can only initialize an entity of type pointer.

NUMBER OF PARAMETERS IN CALL DOES NOT MATCH DECLARATION

A procedure or function may be executed by being called only with exactly as many parameters as it was declared with.

PARAMETER LIST NOT PRECEDED BY FUNCTION NAME

While processing an expression, a parameter list (a list of expressions enclosed in parentheses) was found. However, the entity preceding the parameter list was not of type function.

PATTERN VARIABLES MAY NOT BE PREDICATE OR FUNCTION SYMBOLS

A variable name appearing in a PATTERN statement in a rulefile was found in a context where it would make a predicate or function into a pattern. This is a second-order match, and is prohibited by the prover. Rulefile predicates and functions must be constants; they cannot be instantiated in the prover.

PROCEDURE NAME EXPECTED

You had a statement which looked like a procedure call, but the entity that should be the name of the procedure was not found or was declared to be something else.

PROCEDURE OR FUNCTION DECLARED FORWARD AND NOT FOUND

You declared a function or procedure to be FORWARD and then didn't provide the body of the function or procedure. If you just want to specify the properties of a function or procedure without specifying its body, use EXTERNAL or EXTERN instead of FORWARD.

PROCEDURE OR FUNCTION DELCARED FORWARD AND RESPECIFIED

When the body of a procedure or function declared forward appears, the parameter list, type, initial, entry, exit, and global portions are not duplicated. The format is "PROCEDURE or FUNCTION <identifier> ; <block>".

RECORD FIELD MODIFIES ENTITY NOT OF TYPE RECORD

Following an entity, the notation ".<identifier>" was found, as if the entity was a record of which a particular field was being selected. However, the entity being so modified was not of type RECORD.

RECORD FIELD NAME MAY NOT BE USED AS VARIABLE (vcg)

You have used a record field name that is the same as the name of a variable in your program. This is not permitted.

REFERENCE CLASS EXPECTED

Processing an assertion in Pascal code, a term of the form "•<identifier1> ⊂ <identifier2> ⊃" was found. The entity •<identifier1> was not the name of a reference class known by the parser. You need to declare a type that is a ↑ <identifier1> to get the reference class.

SCHEDULER MAY NOT BE SCHEDULED

A scheduler is used to control access to modules, and is assumed to run in a hardware mutual exclusion state. As such, to have a scheduler for a scheduler is a built-in deadlock. Therefore, a syntax error is given.

SCHEDULER NOT DECLARED OR OF WRONG TYPE

A scheduler for a module must be of type scheduler. Your name wasn't. Alternatively, you tried to enter some condition variables and didn't have a scheduler with a RECEIVES field (concurrent version only).

SUBSCRIPT TYPE DOES NOT MATCH SUBSCRIPT DECLARATION

Each subscript of an array must be of a compatible type with the declaration of that array. Your use of an array did not match on one or more of its subscripts. The printout may tell you the type expected.

SUBSCRIPT TYPE MUST BE SCALAR

When defining an array type, the type of each subscript may not be a record, pointer, array, or file.

SYMBOL TABLE TOO OLD -- PLEASE RECREATE IT

The LOADSYMBOLS and DUMPSYMBOLS operations have an internal check which make sure that they are consistently used. You have tried to do a LOADSYMBOLS operation using a file that was created too long ago -- there has been an incompatible change in the verifier symbol table structure since then. You must recreate the file by another DUMPSYMBOLS operation or get an older verifier.

THIS BUILT-IN FUNCTION MAY NOT BE QUALIFIED

You tried to follow a built-in function call by additional characters. Most built-in functions, such as TAG or EOF, may not be qualified by de-referencing, record fields, subscripts, or union selection.

THIS ITEM MAY NOT BE USED IN RULEFILES

Currently not used, it may be adopted when type checking is extended in rulefiles.

THIS PROCEDURE NOT FOUND ON YOUR SYMBOL TABLE FILE

The LOADSYMBOLS operation has gone through the entire symbol table file you gave it and did not find the environment of the procedure or function you specified.

TOO MANY CONDITIONS IN THIS CLASS--CHANGE CVS

The maximum number of condition variables which may appear within any class is determined by the value of a constant named CVS which must appear in your program. CVS did, in fact, appear, but you tried to declare a class containing more condition variables (concurrent version only).

TYPE ERROR IN DATA TRIPLE

In a data triple appearing in a program assertion, the second entity of the data triple must be of a correct type to qualify the first entity. The third entity must be of a type which can be stored in an element of the first entity.

UNDEFINED OR UNKNOWN RECORD FIELD

You tried to qualify an entity of record type with a record field which did not appear in the declaration of that type.

UNDEFINED OR UNKNOWN UNION TYPE FIELD

You tried to qualify an entity of union type with a union field which did not appear in the declaration of that type.

UNION FIELD MODIFIES ENTITY NOT OF TYPE UNION

You tried to modify an expression not of union type with a union field.

UNKNOWN ERROR MESSAGE -- PARSER OR VCG ERROR

An attempt was made to emit an error message from within the parser or VCG. However, that message did not exist on the error message file. Please let someone who fixes things know!

VARIABLE IN WITH-STMT NOT OF TYPE RECORD

The expressions following the keyword WITH must each evaluate to be a variable of type RECORD.

VARIABLE MUST APPEAR IN GLOBAL STATEMENT

Within a procedure, you tried to reference a global variable which did not appear in a GLOBAL statement. Globals may be referenced within functions without appearing in a GLOBAL statement; indeed this statement is prohibited within functions. See the next error for further discussion.

VARIABLE MUST APPEAR IN GLOBAL STATEMENT PRECEDED BY VAR

You have tried to change the value of a global variable. When you do this in a procedure, you must put the name of the variable (or reference class, for pointer changes) into a GLOBAL statement VAR list. The VAR list is necessary only when values are changed; not merely referenced. If the global is merely referenced, it need not be preceded by VAR (and will simplify proof problems for you if it isn't). GLOBAL statements are not permitted in functions; in that case you may have to convert the function into a procedure which returns its value as a VAR parameter.

A global variable appearing in an INITIAL statement must also appear in a GLOBAL statement. The assumption is that changing the value of the global is intended; if the global is not changed, merely use the global name in assertions and drop the INITIAL statement.

Note that reference classes of pointer types may be globals, and thus may have to appear in the GLOBAL statement.

VAR PARAMETER MAY NOT HAVE EXPRESSION PASSED TO IT

You have tried to pass an expression to a procedure in a position where a VAR parameter was declared. This is not permitted, as it is not defined what it means to store into such an entity in Pascal. You can pass an expression to a non-VAR parameter, but of course, such expression will be strictly an input value to the procedure. Note also that GLOBAL statements are not permitted in functions, which may not have side effects. Thus, getting this syntax error within a function can require re-writing the function as a procedure.

VISIBLE BASE TYPE NOT DEFINED WITHIN MODULE

A type name appearing in a BASETYPE statement must be fully specified within the module. It must appear in a normal TYPE statement therein.

WAIT_FOR STMT REQUIRES CONDITION VAR AS PARAMETER

To use a wait_for statement, there can be only one parameter. It must be declared as a condition variable within the class (concurrent version only).

WAIT_FOR STMT REQUIRES APPROPRIATE DECLARATION WITHIN SCHEDULER

To use a wait_for statement, there must be a scheduler containing a procedure named wait_for. Further, that scheduler must have exactly two parameters: the first of type CVLINK, the second of type SCHEDPROCNAME (concurrent version only).

Appendix C Verification Condition Generator

Notation

x_0 is a fresh identifier
@t is a reference class

C.1 Assertion statements

ASSERT

$$\frac{P \supset L,\ L \supset R}{P\{\text{ASSERT } L\}R}$$

ASSUME

$$\frac{P \wedge Q \supset R}{P\{\text{ASSUME } Q\}R}$$

COMMENT

$$\frac{P \supset (Q \wedge Q \supset R)}{P\{\text{COMMENT } Q\}R}$$

C.2 Basic executable statements

ASSIGNMENT

$P|^{x}_{e}\ \{x:=e\}\ P$ (where x is an identifier)

$x_0=\langle x,.f,e\rangle \supset P|^{x}_{x_0}\ \{x.f:=e\}\ P$ (where x is a record)

$a_0=\langle a,[i],e\rangle \supset P|^{a}_{a_0}\ \{a[i]:=e\}\ P$ (where a is an array)

$@t_0=\langle @t,\subset x\supset,e\rangle \supset P|^{@t}_{@t_0}\ \{x\uparrow:=e\}\ P$ (where x↑ has type t)

CASE

$$\frac{\text{for } i=1,\ldots,n,\ Q\{\text{ASSUME } c=e_i;\ S_i\}R,\quad c=e_1 \vee \ldots \vee c=e_n}{Q\{\text{CASE } c \text{ OF } e_1:S_1;\ \ldots;\ e_n:S_n\}R}$$

The precondition $c=e_1 \vee \ldots \vee c=e_n$ is omitted if c has a subrange type containing only $e_1, \ldots, e_n$.

CONDITIONAL

$$\frac{Q\{\text{ASSUME } L;\ B\}R,\ Q\{\text{ASSUME } \neg L;\ C\}R}{Q\{\text{IF } L \text{ THEN } B \text{ ELSE } C\}R}$$

GOTO and Labels

The verifier does not permit a block to be exited by a non-local GOTO. The other restriction is that every closed path formed by GOTOs and labels must contain an ASSERT statement. Each path through a labelled statement produces a separate verification condition. The rule used by the verifier constructs an assertion at each label. In the general case, it is somewhat complicated. However, if a label is at an ASSERT statement, the rule for GOTO is

$$\frac{P \supset R_j}{P\ \{\text{GOTO } j\}\ Q}$$

where the statement at label j is ASSERT R_j.

NEW

There are two axioms for the NEW(x) statement. The first axiom applies if x is an identifier. Otherwise, the second axiom is used.

$$\neg\text{POINTER_TO}(x_0,\alpha t) \wedge x_0 \neq \text{NIL} \supset Q^{\alpha t}_{\alpha t[x_0]}{}^{x}_{x_0}\ \{\text{NEW}(x)\}\ Q$$

$$\frac{Q\{\text{NEW}(s_0);\ s:=s_0\}R}{Q\{\text{NEW}(s)\}R}$$

REPEAT

REPEAT statements are translated into equivalent WHILE statements. As part of this translation, labels appearing in the body are automatically renamed.

WHILE

$$\frac{I \wedge L\{B\}I,\ P \supset I \wedge \forall \delta_B(I \wedge \neg L \supset Q)}{P\{\text{INVARIANT I WHILE L DO B}\}Q}$$

where δ_B is the set of variables changed in B.

WITH

WITH statements are eliminated by translation.

C.3 Procedures and functions

PROCEDURES

A Procedure declaration has the form:

```
PROCEDURE p(U; VAR V)
GLOBAL (G; VAR H);
INITIAL X=X0;
ENTRY I(U,G,V,H);
EXIT  O(U,G,V,H,X0);
BEGIN
  body
END;
```

where

U is the set of formal value parameters
V is the set of formal variable parameters
G is the set of unchanged global variables
H is the set of changed global variables
X0 is a set of logical variables that may appear in assertions.

Two rules are used to define the semantics of procedures: The procedure declaration rule is used to check the consistency of the assertions in the declaration. The procedure call rule is used to check the consistency of programs that call p.

There is a slight complication in the declaration rule concerning value parameters whose values can be changed by the body. If a procedure q calls p with a value parameter, p operates on a copy of the value, so if p changes the value of its parameter, the change is not visible to q. In the procedure declaration rule, this behavior is modelled by requiring the exit assertion to refer to the initial values of value parameters, before execution of the body. The value parameters U are divided into the subsets U_v of variables that can be changed in the body, and U_c, variables that remain constant. New variables $U0_v$ are introduced to stand for the initial values of value parameters that can be changed in the body. Occurrences of variables in U_v in the exit assertion are replaced by the new variable in $U0_v$, to insure that the exit assertion refers to only the initial value.

The declaration rule checks the consistency of a procedure with its ENTRY and EXIT assertions by proving the formula

$$I(U_c,U_v,G,V,H) \wedge X=X0 \wedge U_v=U0_v \ \{body\} \ O(U_c,U0_v,G,V,H).$$

In the procedure call rule below, A is the set of actual value parameters, and B is the set of actual VAR parameters. Each VAR parameter consists of an identifier, β_i, followed by a possibly empty sequence of component selectors, s_i. The call rule introduces new variables $t_1, \ldots, t_n$ to save the values of the selector sequences of the VAR parameters. B_0 is the set $\{\beta_1_0, \ldots, \beta_n_0\}$ of new variables introduced to stand for the values of the VAR parameters after the procedure call. Similarly, H_0 is a set $\{h_1_0, \ldots, h_m_0\}$ of new variables for the VAR globals. The variables $actuals_X$ are the actual initial values corresponding to the formal variables in X0.

The formula Φ_p asserts that the final value of each variable changed by the procedure call is functionally dependent on the initial values of all the parameters. A new uninterpreted function symbol, $p_i(A,G,B,H)$, is introduced to stand for the final value of each VAR parameter and VAR global.

$$\frac{\begin{array}{l} Q\ \{t_1 \leftarrow s_1; \ldots; t_n \leftarrow s_n\}\ (I(A,G,B,H) \wedge \\ \quad (O(A,G,B_0,H_0,actuals_X) \wedge \Phi_p(A,G,B,H,B_0,H_0) \\ \qquad \{\beta_1 \leftarrow <\beta_1,t_1,\beta_1_0>; \ldots; \beta_n \leftarrow <\beta_n,t_n,\beta_n_0>;\ H \leftarrow H_0\}\ R)) \end{array}}{Q\ \{p(A,B)\}\ R}$$

where $\Phi_p(A,G,B,H,B_0,H_0) =$

$$\beta_1_0=p_1(A,G,B,H) \wedge \ldots \wedge \beta_n_0=p_n(A,G,B,H)$$
$$\wedge\ h_1_0=p_{n+1}(A,G,B,H) \wedge \ldots \wedge h_m_0=p_{n+m}(A,G,B,H).$$

Example: consider the declaration

```
PROCEDURE p(d:m1; VAR e:m2; VAR f:m3);
INITIAL d=d0,f=f0;
ENTRY I(d,e,f);
EXIT  O(d,e,f,d0,f0);
BEGIN
  body
END;
```

Then for the call p(a,b[i],x↑.f), we have:

$$Q\ \{t_1 \leftarrow selector([i]);\ t_2 \leftarrow selector(\text{⊂}x\text{⊃}.f)\}$$
$$(I(a,b[i],\text{⊂}x\text{⊃}.f) \wedge$$
$$(O(a,b_0,\text{⊂}t_0,a,\text{⊂}x\text{⊃}.f) \wedge b_0 = p_1(a,b[i],\text{⊂}x\text{⊃}.f) \wedge \text{⊂}t_0 = p_2(a,b[i],\text{⊂}x\text{⊃}.f)$$
$$\{b \leftarrow \langle b,t_1,b_0\rangle;\ \text{⊂}t \leftarrow \langle \text{⊂}t,t_2,\text{⊂}t_0\rangle\}\ R(a,b,x,\text{⊂}t))$$

$$Q\ \{p(a,b[i],x\uparrow.f)\}\ R(a,b,x,\text{⊂}t).$$

The assignment rule reduces the upper formula to

$$Q \supset (I(a,b[i],\text{⊂}x\text{⊃}.f)$$
$$\wedge\ (O(a,b_0,\text{⊂}t_0,a,\text{⊂}x\text{⊃}.f) \wedge b_0 = p_1(a,b[i],\text{⊂}x\text{⊃}.f) \wedge \text{⊂}t_0 = p_2(a,b[i],\text{⊂}x\text{⊃}.f)$$
$$\supset R(a,\langle b,[i],b_0\rangle,x,\langle \text{⊂}t,\text{⊂}x\text{⊃}.f,\text{⊂}t_0\rangle))).$$

FUNCTIONS

A Function declaration has the form

```
FUNCTION f(U):m;
ENTRY I(U);
EXIT O(U,f);
BEGIN
  body
END;
```

where U is the set of formal value parameters.

The body contains assignment statements of the form f:=e, which assign a value to the function. Occurrences of f as a term in the exit assertion O are interpreted to stand for the value of the function. When f appears as a function sign in O, it has its usual interpretation -- the function f.

The system checks the consistency of a function declaration by proving the formula

$$I(U)\ \{body|^{f:=e}_{f_fn:=e}\}\ O(U,f_fn).$$

A new variable, f_fn, is introduced and the assignment statements are renamed, to avoid conflicts between the two interpretations of f. The formula above is used when none of the varables in U can be changed by the body. When this is not the case, additional new variables are introduced as in the procedure declaration rule.

The semantics of function calls are not given by a single rule. Instead, the semantics of the executable Pascal statements have been defined to account for function calls. To simplify the presentation, the axioms stated elsewhere in this appendix assume that no function calls occur in executable statements. Thus the actual rules implemented in the system are slightly more complex than the ones listed here.

To indicate the general approach, consider assignment statements x[i]:=j, where i and j are expressions containing function calls. Let $f_1(A_1), \ldots, f_n(A_n)$ be an order in which the function calls can be evaluated to execute the assignment, and let $I_k(U_k)$ and $O_k(U_k,f_k)$ be the entry and exit assertions for f_k. Then under the actual axiom used in the system, the conditions for assignment are expressed by

$$I_1(A_1) \wedge (O_1(A_1,f_1(A_1)) \supset \ldots\ I_n(A_n) \wedge (O_n(A_n,f_n(A_n)) \supset R|^{x}_{<x,[i],j>}) \ldots)\ \{x[i]:=j\}\ R.$$