Middle Gypsy 2.05 Definition

Donald I. Good
Ann E. Siebert
William D. Young

Technical Report 59

15 May 1990

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

Abstract

This report contains the mathematical definition of the Middle Gypsy 2.05 language. Gypsy 2.05 is a language for describing computer programs. Middle Gypsy is a slightly modified subset of Gypsy 2.05. The subset excludes concurrency and type abstraction.

The mathematical definition of Middle Gypsy 2.05 consists of three parts, a context-free grammar and two mathematical functions. The syntax of the language is defined by the grammar. The grammar is presented in a form which can be processed by the parser generator Yacc and the lexical analyzer Lex. The semantics of the language is defined by the two mathematical functions. One function defines the part of Gypsy which describes mathematical functions. The other defines the part which describes the operation of computer programs. Both functions are defined in the Boyer-Moore logic.

Acknowledgment

This work was sponsored in part at Computational Logic, Inc. by the National Computer Security Center (Contract MDA904-89-C-6010). The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the National Computer Security Center or the U.S. Government.

Preface

This report describes a mathematical definition of Middle Gypsy 2.05. The definition itself is contained in Appendices A and D. The other parts of this report describe that definition.

The standard definition of Gypsy 2.05 presently is the Gypsy 2.05 report [Good 89a]. Because of the precision and rigor of a mathematical definition, it is highly desirable that the current Gypsy 2.05 standard definition be replaced by a mathematical one. The definition contained in this report is put forward as a *working* standard definition for the Middle Gypsy part of the Gypsy 2.05 language.

The mathematical definition is a formal specification of the Middle Gypsy 2.05 language, and it consists of about 200 pages of functions defined in the Boyer-Moore logic. This definition has the precision of formal mathematics, and this gives Middle Gypsy 2.05 a precise interpretation. There are no questions about what the definition says (as there are in the Gypsy report). Further, because the definition has been admitted to the Boyer-Moore logic, we know what it says is mathematically consistent. But there remains the important issue of whether those 200 pages of mathematics say the "right" thing; and ultimately, this must remain a subjective decision.

It is for this reason that the current definition is put forward as a working standard. The definition should be rigorously reviewed and evaluated before being adopted as *the* standard definition of Middle Gypsy 2.05. It is recommended that this review and evaluation have (at least) the following two elements.

First, a rigorous analysis should be made of the agreement between the working standard and the Gypsy Verification Environment (GVE). The GVE is *not* a definition of the language, and it should not be interpreted as such. It is a tool for analyzing sentences in the language, and that analysis should comply with the definition. But of necessity, the GVE must deal with many specific details of the language. Discovering items of disagreement between the GVE and the working standard would reveal items of disagreement that should be resolved. We believe that this resolution process would be a healthy one for both the working standard and the GVE, and it is likely that both the GVE and the working standard would evolve as a result of this process.

Second, because the definition is in rigorous mathematical form, it is possible to state and prove invariant properties about the definition. Many of these properties were discovered in the course of developing the definition, and some of them are mentioned in this report. However, rigorous formulation and proof of these properties was not possible within the scope of this effort. However, doing these proofs would have increased substantially our belief that the 200 pages of Boyer-Moore logic does say the "right" thing.

Chapter 1
PROGRAM DESCRIPTIONS

Gypsy is a language for describing computer programs. A *language* is a set of sentences; a *sentence* is a sequence of symbols; and in the Gypsy language, a *symbol* is a sequence of ASCII characters. A *computer program* is a control mechanism which controls the operation of a computer. Thus, a sentence in the Gypsy language is a sequence of ASCII characters which describes computer control mechanisms.

A Gypsy sentence can describe three kinds of things about computer programs.

- It can describe the composition of control mechanisms in the style of a traditional von Neumann programming language.
- It can describe operating constraints on those mechanisms.
- It can describe the mathematical functions from which the operating constraints are composed.

Although the primary purpose of Gypsy is to describe computer programs, it is not required that every Gypsy sentence *must* describe *some* program. Therefore, a Gypsy sentence *may* be used just to describe mathematical functions.

1.1 Computer Programs

A Gypsy sentence can describe the composition of a computer program. A *computer program* is a mechanism which controls the sequence of operations performed by a computer. The effect of an operation is to change the state of the computer. Thus, the effect of the mechanism is to produce a sequence of states.

One can think of abstract, conceptual control mechanisms as well as physical ones. But, to keep conceptual and physical issues sharply distinguished, in this report a computer program will be regarded purely as a physical mechanism. A program consists of a physically connected collection of two-state devices in a digital computer. The physical state of these devices controls the sequence of state-changing operations which are performed by the computer.

Figure 1-1 shows an example of a sequence of Gypsy symbols which describes a control mechanism which will cause a computer to perform a sequence of operations which computes factorial. The symbols "**`x`**", "**`y`**" and "**`k`**" describe parts of the state of the computer, and other symbols such as "**`;`**", "**`if`**" and "**`loop`**" describe the sequence of operations which are to be performed.

It is important at the outset to distinguish clearly between a physical mechanism and an abstract,

Figure 1-1: Factorial Program

```
type mint = integer[0..100];

procedure fact_prg(var y:mint; x:mint) =
begin
  var k:mint;
  y := 1;
  k := 0;
  loop
    if k ge x then leave
    else k := k + 1;
         y := y * k;
    end;
  end;
end;
```

conceptual model of it. The control mechanism itself is composed of physically connected two-state devices in the computer. The symbols in Figure 1-1 describe an abstract, conceptual model of a control mechanism; and this abstract model describes the physical mechanism. Many details of physical operation are omitted quite intentionally from the conceptual model.

The benefit of a conceptual model is that its operation can be analyzed and used to forecast the behavior of the physical mechanism without actually operating it. The risk of a conceptual model is that it may not be a sufficiently *detailed* or *accurate* model of physical behavior to provide an accurate forecast. This is the normal risk-benefit trade-off associated with any kind of conceptual modeling of physical behavior.

These modeling issues are beyond the scope of this report. They are discussed further in [Good 89b]. The operation of mechanisms which is discussed in the remainder of this report is the operation of the conceptual mechanism which is described by a Gypsy sentence, not the operation which is actually performed by a physical mechanism.

1.2 Operating Constraints

A Gypsy sentence also may contain symbols which state *operating constraints* for a program mechanism. A Gypsy operating constraint states a property which (the conceptual model of) the operation of a program mechanism is *intended* to have. It is a statement of intention. The appearance of an operating constraint in a Gypsy sentence is *not* sufficient to ensure that the operation of the mechanism *does* have the property. To ensure that, mathematical analysis is required.

Figure 1-2 shows the factorial program in Figure 1-1 with some operating constraints inserted. They are the lines beginning with the symbols "**entry**", "**exit**" and "**assert**". In this example, the symbols "**x**", "**y**" and "**k**" in these constraints refer to components of the program state. The symbol "**ge**" in the **entry** constraint and the symbol "**fact_fcn**" in the **exit** and the **assert** constraint refer to mathematical functions on the integers.

Figure 1-2: Factorial Program with Operating Constraints

```
type mint = integer[0..100];

procedure fact_prg(var y:mint; x:mint) =
begin
entry x ge 0;
exit  y = fact_fcn(x);
  var k:mint;
  y := 1;
  k := 0;
  loop
    assert y = fact_fcn(k) & k in [0..x];
    if k ge x then leave
    else k := k + 1;
         y := y * k;
    end;
  end;
end;
```

1.3 Mathematical Functions

Operating constraints for programs are stated in terms of mathematical functions. Therefore, a Gypsy sentence also may contain symbols which describe mathematical functions.

A *mathematical function* is a conceptual correspondence between the elements of two sets of elements, a domain set and a range set. The traditional definition of a function is a set of ordered pairs **`[x,y]`** such that every **`x`** is an element of the domain set, every **`y`** is an element of the range set, and every **`x`** in the set of pairs is unique.

Figure 1-3 shows the Gypsy definition of the mathematical function **`fact_fcn`** which is used in the **`assert`** and **`exit`** constraints in Figure 1-2. A mathematical function may be described in Gypsy by giving a complete definition of it, as illustrated in this figure; or a function may be described just by stating some of its properties. This may be done either with an **`exit`** specification on the function or with a Gypsy **`lemma`**.

Figure 1-3: Factorial Function

```
type mint = integer[0..100];

function fact_fcn(x:mint):mint =
begin
entry x ge 0;
exit  result = if x=0 then 1 else x*fact_fcn(x-1) fi;
end;
```

A mathematical function is a correspondence between two sets. A computer program is a control mechanism. Gypsy uses the correspondence of mathematical functions to state constraints on the operation of control mechanisms. The correspondence and the mechanism are quite different concepts.

1.4 Executable Functions

Gypsy sometimes does use the same symbol to refer to both a program mechanism and a mathematical function. This combination of mechanism and function is known as an *executable function*.

The Gypsy standard functions and operators are executable functions. The **ge** operator in Figure 1-2 is an example. When it appears in the **entry** constraint of the program **fact_prg**, it refers to a mathematical function named **ge**. When it appears in the **if** statement in the program, it refers to a mechanism named **ge**. The Gypsy language uses the same symbol for both the **ge** function and the **ge** mechanism because they are defined so that the function does accurately describe the mechanism.

Figure 1-4: Executable Factorial Function

```
type mint = integer[0..100];

function factorial(x:mint):mint =
begin
entry x ge 0;
exit  result = if x=0 then 1 else x*factorial(x-1) fi;
  var k:mint;
  result := 1;
  k := 0;
  loop
    assert result = factorial(k) & k in [0..x];
    if k ge x then leave
    else k := k + 1;
         result := result * k;
    end;
  end;
end;
```

New executable functions also can be declared in a Gypsy sentence. Figure 1-4 shows how the constrained program **fact_prg** in Figure 1-2 and the mathematical function **fact_fcn** in Figure 1-3 can be combined into a single, executable function **factorial**.

The **entry** and **exit** of an executable function serve a dual purpose. For the function, they state intended mathematical properties. For the program, they state intended operating constraints. *If* the stated mathematical properties do define a function, *and* the program does satisfy the stated operating constraints, *then* the mathematical function does accurately describe the program. The standard executable functions of Gypsy are defined so that they meet these requirements. Whether they are met for executable functions which are declared in a Gypsy sentence depends on how they are declared.

1.5 Middle Gypsy 2.05

By definition, Middle Gypsy 2.05 is the language defined in this report. Middle Gypsy 2.05 differs only very slightly from the language which is described in Chapters 1-9 of the Gypsy 2.05 report [Good 89a]. These chapters do not include concurrency or type abstraction. Hereafter, these two languages will be referred to simply as "Middle Gypsy" and "Gypsy."

Except as noted below, Middle Gypsy is the same language as Gypsy.

- The starting symbol of the Middle Gypsy grammar is **program_description** which generates a sequence of **scope_declaration** phrases, whereas the Gypsy grammar begins with **scope_declaration**.

- The right associative operator **:>** has been placed in a precedence class by itself.
- The **ALL** and **SOME** operators have been restricted to bounded quantification. This restriction was necessary to define them in the Boyer-Moore logic. The bound variables are *not* required to be unique within a Gypsy unit. This requirement was relaxed as a convenience.
- Concurrency (Chapter 10) is not included.
- Type abstraction (Chapter 11) is not included.

The treatment of syntax and semantic errors by the mathematical definition differs in two important ways from what might be expected from reading the Gypsy report. First, the mathematical definition provides a Gypsy interpretation for *all* sequences of symbols, even if they do contain syntax or semantic errors. The interpretation detects and reports these errors. However, the interpretation detects only those which actually occur during a particular interpretation. For example, the semantic error in **5&6** in the expression

```
if x>0 then 5&6 else 4 fi
```

is detected only for interpretations for which **x>0**, and if there is no interpretation for which this is true, then the semantic error in **5&6** would never be detected.

Second, the mathematical definition does not distinguish between static and dynamic semantic errors. Both kinds are detected by the interpretation of a Gypsy sentence, but the definition is not structured as a static analysis phase followed by a dynamic interpretation. Gypsy is designed so that many kinds of semantic errors *can* be detected by a static analysis phase, but the definition does not *require* this. The mathematical definition is structured quite intentionally to specify *what* the semantic errors are but not to specify *how* they are detected. That is left to the discretion of various methods and tools which use the mathematical definition to analyze the interpretation of a Gypsy sentence. (Although Gypsy semantic errors traditionally have been thought of as being either static or dynamic, the Gypsy report actually does not require this distinction.)

The Gypsy Verification Environment (GVE) [Good 88] is an example of a particular analysis tool which has made a particular set of decisions about which semantic errors are to be detected statically and which are to be detected dynamically. However, these decisions are *not* imposed on the mathematical definition. Other analysis tools may quite appropriately make other decisions. Although the behavior of the GVE has been consulted frequently in developing the mathematical definition, the GVE is *not* the definition of the Gypsy language. It is a tool which uses a definition of the Gypsy language to analyze a Gypsy sentence for some very specific kinds of properties. The definition which the GVE has used is the one contained in the Gypsy 2.05 report.

Although Middle Gypsy differs only slightly from Gypsy without concurrency and type abstraction, the *explanation* of Middle Gypsy is quite different from the explanation of Gypsy. In [Good 89a], Gypsy is explained from a procedural point of view in the style of a programming language manual. Middle Gypsy is explained in terms of computer programs, operating constraints and mathematical functions as discussed in Chapter 1.

This new explanation reflects a deeper and more precise understanding of Gypsy, and it is only fair to say that this understanding did not exist at the time Gypsy originally was developed. Much of it has been acquired over approximately 15 years of developing Gypsy, building tools for it and using it in a wide variety of applications. Some of it has come from developing the mathematical definition which is contained in this report.

The structure of the remainder of this report follows the general structure of the mathematical definition of Middle Gypsy. Chapter 2 explains the methods for defining the language, and the remaining chapters explain the Middle Gypsy definition. Chapter 3 discusses the library of Gypsy units which is declared by a Gypsy sentence. Chapter 4 discusses Gypsy types. Chapters 5 and 6, respectively, discuss the mathematical and the procedural interpretation of a Gypsy sentence.

The full mathematical definition of Middle Gypsy is given in Appendices A and D of this report. Chapters 3-6 explain the key ideas of the mathematical definition, but they do not attempt to replicate the definition in prose. These chapters are written with the expectation that the reader will refer frequently to the actual definition given in the appendices.

Chapter 2

MATHEMATICAL DEFINITION

The mathematical definition of Middle Gypsy consists of three parts -- a context-free grammar which defines a set of sentences and two functions **F** and **P** which define the interpretation of those sentences. **F** interprets the parts of those sentences which describe mathematical functions. **P** interprets the parts which describe program operation. Some parts of a Gypsy sentence have just a mathematical interpretation. Some have just a procedural interpretation. Some have both.

Much of the the procedural interpretation by **P** is defined in terms of the mathematical interpretation provided by **F**. For example, **P** uses **F** to interpret the operating constraints of a program.

2.1 Syntax

The syntax of Middle Gypsy is defined by a context-free grammar. The productions of the grammar are defined in the Yacc input file in Appendix A.3. (See [Aho 86]). The tokens of the grammar are defined in the Lex input file in Appendix A.2.

The Middle Gypsy grammar has been derived from the Gypsy grammar in the following steps:

1. The productions of the Gypsy grammar have been rewritten in a form acceptable to Yacc. Nonterminal symbols such as **`<scope declaration>`** have been replaced by **`scope_declaration`**, and the symbol"**`::=`**" has been replaced by a single colon. Gypsy productions with the **`[]`** and **`{}`** notation have been replaced by equivalent productions with the **`empty`** production. In the Middle Gypsy productions, lower case is used for nonterminal symbols and upper case is used for tokens. The literal terminal symbols in the Gypsy productions have been replaced by the Lex name of the corresponding token -- e.g., "**`;`**" has been replaced by **`SEMI_COLON`**.

2. Several of the Gypsy productions have been eliminated by replacing every occurrence of their left-hand side by their right-hand side throughout the grammar. In the Gypsy report, many of these were used to carry connotations of semantic checks. In the mathematical definition, these checks are done by the interpreter functions **F** and **P**. Eliminating these productions from the grammar makes the mathematical definition more clear and concise.

3. The starting symbol for the Middle Gypsy grammar is **program_description** instead of **scope_declaration**.

```
program_description : scope_declaration_list opt_semi_colon

scope_declaration_list :
       scope_declaration
     | scope_declaration_list SEMI_COLON scope_declaration

opt_semi_colon :
       empty | SEMI_COLON

empty :
```

4. The productions in the Gypsy report which involve concurrency and data abstraction have been eliminated.
5. The grammar for **expression** has been reformulated because the one in the Gypsy report is ambiguous.
6. The right associative operator **:>** has been placed in a precedence class by itself because Yacc does not allow a single precedence class to contain both left and right associative operators.

2.2 Semantics

A Middle Gypsy sentence **x** is a sequence of ASCII characters. The Middle Gypsy language is the set of sequences of ASCII characters which is generated by the grammar in Appendix A from the starting symbol **program_description**. The semantics of a sentence **x** is defined by two functions, **F** and **P**, which interpret character sequences. To distinguish these functions from those which are described *by* the sentence **x**, the functions **F** and **P** are referred to as *meta-functions*. The definitions of these functions in the Boyer-Moore logic [Boyer & Moore 79, Boyer & Moore 88] are given in Appendices D.1 and D.2. The pages on which they are defined are given in the index of this report. Because the Boyer-Moore logic will not allow a function named **F** to be defined, the functions **F** and **P** are named **meta_F** and **meta_P** in Appendix D.

The meta-functions **F** and **P** are defined on character sequences. They are defined in terms of two additional meta-functions **gF** and **gP** which are defined on parse trees generated by the grammar. The meta-functions **gF** and **gP** are defined in terms of these parse trees. These two functions make a rigorous connection between the syntax and the semantics of Middle Gypsy. To maintain consistency between the representation of the production rules in the Boyer-Moore logic and their representation in Yacc form, a program has been written which extracts the grammar productions from the definitions of the meta-functions in Appendix D. The rules extracted by this program have been manually compared with the syntax in Appendix A.3.

2.2.1 Interpreter Functions

The meta-function **F(e,c,v,n,x)** interprets a sequence of characters **e** (which appears in scope **c** in a Gypsy sentence **x**) as defining the evaluation of a mathematical expression. The expression **e** contains symbols which refer to constants, free-variables and mathematical operators and functions. The argument **v** is a name-value mapping which associates specific values with the free variables in the expression **e**. The argument **n** is a non-negative integer which ensures that **F** is a total function. The value of **F(e,c,v,n,x)** is the value which results from applying the mathematical functions named in expression **e** to their respective arguments.

The meta-function **P(m,c,s,n,x)** interprets a sequence of characters **m** as describing the operation of a computer program mechanism. The sequence **m** describes a control mechanism in terms of its components. The sequence contains symbols which refer to mechanisms, compositions of mechanisms and to the components of the state on which those mechanisms operate. The argument **s** is a name-value mapping which describes the state which is operated on by the mechanism **m**. The argument **n** is a non-negative integer which ensures that **P** is a total function. The value of **P(m,c,s,n,x)** is the final state which results from operating the mechanism **m** on the initial state **s** for at most **n** steps.

The meta-functions **F** and **P** take similar kinds of arguments. The arguments **e**, **m** and **x** are character sequences. The argument **c** is the (litatom) name of a Gypsy scope. The argument **n** is a number which ensures that **F** and **P** are total functions. The arguments **v** and **s** are name-value mappings.

However, there also are important differences. In **F**, the character sequence **e** is a mathematical expression, and the name-value mapping **v** is an association of specific mathematical values with the names of free variables which appear in **e**. The value of **F** is the result of composing mathematical functions on specific values in the domains of those functions. The environment **v** identifies which of those specific values are used in the composition in place of the free variables which appear in the expression. In **P**, the character sequence **m** describes a mechanism, and the name-value mapping **s** describes the state which is operated on by the mechanism. The value of **P** is the name-value mapping which describes the state which results from operating mechanism **m** on initial state **s**.

P describes the effects of composing computer control mechanisms. **F** describes the effects of composing mathematical functions. The meta-function **F** provides a basis for defining a large part of **P**, and **P** uses **F** to determine if a control mechanism satisfies its operating constraints.

2.2.2 Determinacy Marks

Marked objects are used throughout the definition of **F** and **P** to distinguish between determinate and indeterminate objects. A *marked object* is a **[mark,object]** pair. Marked objects are defined in Appendix D.1 by the Boyer-Moore shell **marked**. (For a full explanation of the shell principle, see [Boyer & Moore 88;p.18ff].)

The determinacy of a marked object is defined by the function **determinate**. A marked object **[mark,object]** is *determinate* iff **mark=nil**. The **mark** part of a marked object is used to report various errors which either **F** or **P** might encounter in their interpretation of a character sequence. These marks report errors which indicate situations in which a determinate interpretation cannot be made.

2.2.3 Indeterminate Interpretations

Both **F** and **P** are defined for all sequences of ASCII characters, even if those sequences are not generated by the Middle Gypsy grammar; and both functions return marked objects. If either **F** or **P** is applied to a character sequence **x** which is not in the set of sentences generated by the grammar, it returns a marked object which reports that **x** has a syntax error.

Semantic errors are reported in the same way. Both **F** and **P** check for semantic errors as they interpret a character sequence. If an error is encountered, it is reported by returning an appropriately marked object.

F and **P** report a semantic error *only if* it actually is encountered in interpreting **x**. A sequence **x** which contains *potential* semantic errors which are *not* actually encountered during interpretation has a determinate interpretation. For example, if **a** is declared to be of type integer, then the mathematical value of **a>0** which is produced by **F** is determinate and the value of **a[1]** is indeterminate. But the value of the expression

```
if a>0 then 4 else a[1] fi
```

is determinate when the value of **a** is greater than zero because, in this case, **F** does not evaluate **a[1]**.

Except as noted, all of the semantic errors of Gypsy which are appropriate to Middle Gypsy are included in the mathematical definition. However, the Middle Gypsy definition does not distinguish between static and dynamic semantic errors. "Static" or "dynamic" refers to *how* a semantic error can be detected. The definition specifies *what* the semantic errors are, but it does not specify *how* they are to be detected.

2.2.4 Parse Trees

The meta-functions **F** and **P** are defined on character sequences. To connect these functions clearly to the grammar, they are defined in terms of two similar functions on parse trees. Because the grammar is unambiguous, there exists a parsing function **pT(x,g)** which maps the character sequence **x** and the goal symbol **g** into the unique parse tree which describes the derivation of **x** from **g**. (If a sentence **x** is not generated from **g**, then **pT(x,g)=nil**).

The function **pT** is not defined in the Boyer-Moore logic. The acceptance of the grammar by the Yacc parser generator as unambiguous is taken as sufficient evidence that such a function exists. However, in the logic, **pT** is constrained so that it either produces a well-formed parse tree or **nil**. The well-formedness of a parse tree is defined by the function **parse_treep**.

The meta-functions **F** and **P** are defined in terms of **pT**, **gF** and **gP** as follows:

```
Let pTe = pT(e,'expression)

    pTm = pT(m,'statement)

    pTx = pT(x,'program_description)

F(e,c,v,n,x) = if pTe=nil
               then marked("Expression Syntax Error", nil)
               else if pTx=nil
                    then marked("Program Description Syntax Error", nil)
                    else gF(pTe,c,v,n,pTx)

P(m,c,s,n,x) = if pTm=nil
               then marked("Statement Syntax Error", nil)
               else if pTx=nil
                    then marked("Program Description Syntax Error", nil)
                    else gP(pTm,c,s,n,pTx)
```

The meta-functions in Appendix D are defined in terms of parse trees which are produced by **pT**. One of these trees is assumed to have litatoms at each of its intermediate nodes which are identical to the symbols in the Yacc grammar. Nonterminal are distinguished from terminal symbols by the function **tokenp**.

The leaves of the tree are the ASCII characters which compose the lexemes of the tokens. The fringe of a parse tree may contain any characters in a Gypsy sentence except white space. Figure 2-1 shows part of **pT(x,'program_description)** for the factorial sentence in Figure 1-4 (The **'** marks are omitted from the litatoms).

Figure 2-1: Partial Parse Tree of Factorial Example

```
                                        program_description
                                                  |
                                    ----------------------------
                                    |                          |
                          scope_declaration_list        opt_semi_colon
                                    |                          |
                            scope_declaration              semi_colon
                                    |                          |
                                    |                          ;
                                    |
   -------------------------------------------------------------------
   |         |        |     |          |                 |        |
 scope  identifier  equal begin   unit_or_name_    opt_semi_colon end
   |         |        |     |    declaration_list        |        |
 -----       -        =   -----        |              semi_colon ---
 |||||       |            |||||        |                 |       |||
 scope       A            begin       ...                ;       end
```

The functions for trees in Appendix D.1 are **`mk_tree, root`** and **`subtrees`**. Parse trees are recognized with the function **`rule`** and the recognizers for the tokens of the grammar.

The function **`rule(u,p)=T`** iff **`u`** is a tree which generates its subtrees from the grammar production **`p`**. A grammar production with left-hand side **`lhs`** and right-hand side **`rhs`** is constructed by the function **`prodn(lhs,rhs)`**. For example, if **`u`** is the tree shown in Figure 2-1, then

```
rule(u,prodn('program_description,
             list('scope_declaration_list, 'opt_semi_colon))) = T
```

This corresponds to the Yacc production

```
program_description : scope_declaration_list opt_semi_colon
```

Every nonterminal node of a parse tree must satisfy **`rule`** for some production of the grammar, and every terminal node must satisfy one of the token recognizer functions **`reserved_wordp, special_symbolp, character_valuep, digit_listp, entry_valuep, identifierp`** or **`string_valuep`**. The function **`parse_treep(u)`** recognizes trees that are generated by the grammar from these rules.

The definition of **`gF`** illustrates how **`rule`** normally is used by the meta-functions. First, **`gF`** uses **`rule`** to find which grammar rule applies to the root of the parse tree of its argument **`e`**. Then it returns the result which is appropriate for that rule. A particular grammar rule normally will appear in several different meta-functions.

The program that extracts the Yacc form of the production rules from meta-functions converts all occurrences of **`prodn(lhs,rhs)`** in Appendix D into **`lhs : rhs`** and eliminates duplicates. It also produces a separate rule for each unary and binary operator. Every grammar rule produced by this extraction program occurs in the grammar in Appendix A.3. The following productions do not occur because they are not used in the meta-functions:

```
opt_group_name : empty | COND

opt_semi_colon : empty | SEMI_COLON

proof_directive : PROVE | ASSUME
```

2.2.5 Tagged Grammar

The occurrences of **`prodn`** in the the definition of a meta-function show the grammar productions which are associated with that function. It also is useful to see how the meta-functions are related to each individual grammar production. This is shown by the tagged grammar in Appendix C. The semantic equations for a production in the tagged grammar show how that production effects each meta-function which refers to the production.

The tagged grammar also is constructed mechanically from the meta-function definitions. The **`tag`** function which appears within **`prodn(lhs,rhs)`** is used only for this mechanical construction. The function **`tag(gsymbol,label)`** produces the symbol **`<gsymbol, label>`** for a production in the tagged grammar, and **`label`** in the meta-function equations for that production refers to the parse tree with root **`gsymbol`**.

From a preliminary investigation, it appears that the tagged grammar is a restricted form of attribute grammar [Aho 86]. However, to avoid possible confusion with established terminology until a more thorough investigation is possible, it is referred to as a "tagged grammar."

Chapter 3
LIBRARY

The Gypsy library contains all scope, type, function, constant, lemma, procedure and name declarations. These are the descriptions of the functions and programs which are interpreted by **gF** and **gP**.

3.1 Scope and Unit Declarations

A Middle Gypsy program description sentence is a sequence of scope declarations; a scope declaration is a sequence of unit or name declarations. The function **scope_list(pd)** is the list (of the parse trees) of all scopes which are declared in a program description sentence **pd**. The function **unit_list(sd)** is the list (of the parse trees) of all units which are defined by a scope declaration **sd**. The Middle Gypsy library consists of all units which are defined in all scopes in a program description sentence.

Every unit declaration **ud** in a scope **sd** defines one or more units, and each of these appears in **unit_list(sd)**. For a function, constant, lemma or procedure declaration **ud**, just one unit is defined, **unit_list(ud)=ud**. A type declaration also defines just a single unit unless it is a scalar type. The declaration of a scalar type also defines a constant unit for each scalar value which appears in the type definition. The constant units are defined by **derived_units(n,u)** where **n** is the name and **u** is the type definition of the scalar type.

Every name declaration **nd** in a scope also defines one or more units. If **nd** has just one local renaming, then **unit_list(nd)** is a list with just one element **nd**. If **nd** has more than one local renaming, **unit_list(nd)** produces a list which has a complete name declaration for every local renaming in **nd**. For example,

```
NAME P, Q FROM S
```

defines the same list of units as

```
NAME P FROM S ; NAME Q FROM S
```

3.2 Resolving Unit References

References to a unit by identifier **i** in a scope named **sn** in a program description sentence **x** are resolved by the function **ref(i,sn,x)**. This function uses **scope_list** and **unit_list** to return a pair **[hn,ud]** where **hn** is the name of the home scope in which the unit is declared and **ud** is the declaration of the unit.

The reference **ref(i,sn,x)** is allowed at most one use of a name declaration. It also checks that

`scope_list(x)` has exactly one scope declaration **`sd`** such that **`scope_name(sd)=sn`** and that there is exactly one unit declaration **`ud`** in **`sd`** such that **`local_name(ud)=i`**. If these conditions are not met, **`ref(i,sn,x)`** reports an error.

Chapter 4
TYPES

A Gypsy *type* specifies a set of values and a distinguished element of that set called the default initial value. A type is identified by a type descriptor, and a typed value is composed of a type descriptor and an untyped value. Gypsy describes both the evaluation of mathematical functions and the operation of computer programs in terms of marked, typed values.

4.1 Type Descriptors

A Gypsy type is identified by a type descriptor. The set of values and the default initial value specified by a type are defined by its type descriptor. The descriptors for the various types are given in Appendix E.

4.1.1 Value Set

A *typed value* is a pair **`[d,u]`** which satisfies the invariant relation **`dtype(d,u)=T`**. The function **`dtype`** interprets **`d`** as a *type descriptor* and **`u`** as an untyped value. If **`dtype(d,u)=T`** then the pair **`[d,u]`** is a member of the value set of the type identified by **`d`**, and if **`dtype(d,u)=F`** then **`[d,u]`** is not a member of the value set. (In addition to **`T`** and **`F`**, **`dtype`** also may return values indicating that **`d`** contains errors or references to pending units.)

The basic constructor function for typed value pairs is **`mk_typed`** with extractors **`type_part`** and **`value_part`**. This constructor, however, does not impose the **`dtype`** invariant. It is imposed on the two parts **`d`** and **`u`** of a typed value by the function **`typed(d,u)`**, and two additional extractor functions are defined

Two additional extractor functions for typed values are **`type`** and **`value`**, and the following relations hold:

```
type([d,u]) = d
value([d,u]) = u
```

The invariant imposed by the constructor **`typed`** is

```
dtype(type(typed(d,u)),value(typed(d,u))) = T
```

If necessary, this invariant is imposed by **`typed(d,u)`** returning the typed value **`[integer_desc, 0]`** where **`integer_desc`** is the descriptor for the standard, unbounded type integer.

The **type** and **value** extractors also can be applied to marked, typed values; in which case, they simply ignore the mark.

```
 type(marked(m, typed(d,u))) =  type(typed(d,u))

value(marked(m, typed(d,u))) = value(typed(d,u))
```

4.1.2 Default Initial Value

The *untyped* default initial value for descriptor **d** is given by the function **udv(d)**, and the following relation holds:

```
not errorp(d) -> dtype(d,udv(d)) = T
```

The *typed* default initial value of the typed value is **[d,udv(d)]**. The standard function **initial** produces this typed value.

4.1.3 Creating Descriptors

Type descriptors are created from type specifications. The type descriptor for a phrase **<type_specification, ts>** which appears in scope **sn** in sentence **x** is created by the function **type_desc(ts,sn,ut,x)**. The type specification may refer to a standard, simple type, or it may refer to a type which is declared with a type declaration. If it does not refer to a standard type, then **type_desc** uses the unit referencing function **ref** (Section 3.2) to create a type descriptor from the type declarations in the Gypsy library. (The **ut** parameter is used to detect recursive types. It is the list of types which already have been used in resolving the reference.)

4.1.4 Structure

Type descriptors are constructed as labeled trees by using the function **mk_tree** (which is also the constructor for parse trees). For each kind of label, a function is defined which gets the value associated with that label. For example, a descriptor **d** for a scalar type has the following form:

```
               'scalar
                  |
  ------------------------------
  |     |     |     |     |    |
'tid  'sid  'crd 'tmin 'tmax 'udv
  |     |     |     |     |    |
```

The functions **tid(d)**, **sid(d)**, **crd(d)**, **tmin(d)**, **tmax(d)** and **udv(d)** get the values associated with those nodes.

The functions **tid(d)** and **sid(d)** are defined in terms of the function **gname(n)** which converts an **<IDENTIFIER, n>** token into a litatom which matches the upper-cased lexeme of the token. Also, it converts the Gypsy identifier **NIL** into the litatom **'NIL~** so that it does not conflict with **NIL** of the Boyer-Moore logic.

The following functions are used uniformly for all descriptors:

tid(d) is **gname(n)** where **n** is the identifier which names the type.

sid(d) is **gname(n)** where **n** is the identifier which names the scope in which the type is declared. (For standard types, **sid(d)=nil**.)

crd(d) is the number of elements in the value set of a scalar type.

tmin(d) is the smallest element in the subrange of a simple type. If there is no subrange, **tmin(d)=nil**.

tmax(d) is the largest element in the subrange of a simple type. If there is no subrange, **tmax(d)=nil**.

max_size(d) is the maximum number of elements allowed for a dynamic type.

selector_td(d)
is the descriptor of the type of the selector for a composed type.

component_td(d)
is the descriptor of the type of the component for a composed type.

udv(d) is the default initial value of the type.

4.1.5 Untyped Values

The untyped values for the scalar types and type integer are integer numbers--ones which are recognized by **integerp** [Bevier 88]. The untyped values for type rational are rational numbers--ones which are recognized by **rationalp** [Wilding 90].

The untyped values for sets and sequence are lists of component elements. The untyped values of array, record and mapping types are lists of **[selector,component]** pairs.

4.1.6 Pending Types

Pending types have a special descriptor which distinguishes them from other types. The descriptor includes the name of the type and the name of the scope in which it is declared. The descriptor for a pending type and the type membership test **in_type** are defined so that an actual parameter of a pending type can be bound to a formal parameter of the same pending type.

4.2 Base Types

Every Gypsy type is identified by some type descriptor **d**. Every Gypsy type also has an associated base type which is identified by the type descriptor **base_type(d)**. In general, **base_type(d)** is the same type descriptor as **d** but without range and size restrictions. The descriptor **base_type(d)** is identical to **d** except that

```
    tmin(base_type(d)) = nil

    tmax(base_type(d)) = nil

max_size(base_type(d)) = nil
```

This gives **base_type(d)** the following properties:

```
base_type(base_type(d)) = base_type(d)

dtype(d,u)=T -> dtype(base_type(d),u)=T
```

4.3 Type Membership

Gypsy functions and procedures take parameters which are typed values. The function which checks the type consistency of actual and formal parameters is the type membership function **`in_type(d,v)`**. If **`in_type(d,v)=T`**, then the *typed* value **`v`** is a member of the type identified by the descriptor **`d`**. The check for type consistency between actual and formal parameters is **`in_type(dformal,vactual)`** where **`dformal`** is the type descriptor of the formal parameter, and **`vactual`** is the typed value of the corresponding actual parameter.

The function **`in_type(d,v)`** is defined so that if **`base_type(d)`** is not equal to **`base_type(type(v))`**, then the type membership test fails (i.e., **`in_type(d,v)=F`**). This defines conditions under which type membership failure can be detected without knowing **`value(v)`**. Gypsy was designed in this way to that these kinds of type membership failures could be detected statically.

Chapter 5

MATHEMATICAL EXPRESSION EVALUATION

The function **`gF(e,c,v,n,x)`** defines the mathematical interpretation of Middle Gypsy. It interprets the sequence of symbols **`e`** as referring to the constants, free variables, bound variables, operators and functions of a mathematical expression; and from these, it evaluates the expression. The constants and free variables of the expression refer to marked, typed values; and **`gF`** applies the operators and functions referred to in the expression to produce the marked, typed value of the expression.

The parameters and the result of **`gF(e,c,v,n,x)`** are as follows:

`e`	is the parse tree of an expression to be evaluated.
`c`	is the (litatom) name of the Gypsy scope in which **`e`** appears.
`v`	is free variable binding environment -- the name-value mapping that maps names of free variables in **`e`** into specific marked, typed values.
`n`	is the maximum depth of Gypsy function calls and quantifiers allowed in evaluating **`e`**.
`x`	is the parse tree of the program description sentence which defines the library containing the declarations of the Gypsy units referred to by **`e`**.
`gF(e,c,v,n,x)`	is the marked, typed value that results from evaluating **`e`**.

The function **`gF(e,c,v,n,x)`** interprets the expression **`e`** as a composition of mathematical operators and functions applied to the constants and free variables in the expression. The definition of **`gF`** defines the interpretation of all Middle Gypsy operators and standard functions which can appear in an expression, and **`gF`** is defined so that these interpretations cannot be changed by any Gypsy unit which may be declared in **`x`**. The sentence **`x`** may contain declarations of functions, and these may be composed from the operators, standard functions and declared functions. But the definition of **`gF`** does not allow these declarations to change the interpretation of the operators and standard functions.

5.1 Constants

The constants of an expression are tokens, and the values they refer to are the following:

```
Token                  Value
-----                  -----

FALSE                  Gfalse
TRUE                   Gtrue

<CHARACTER_VALUE, z>   Gchar(z)

<DIGIT_LIST, z>        minteger(z)

<STRING_VALUE, z>      Gstring_seq(z)
```

5.2 Identifiers

The identifiers in an expression may refer to standard scalar constants (**FALSE, TRUE**), declared scalar constants, free-variables, bound variables, standard functions or declared functions. What value an identifier refers to is determined by the function **Gapply(fn,ap,sn,v,n,x)** where **fn=gname(i)** for the **<IDENTIFIER, i>** token in question.

To determine what kind of thing the identifier **fn** refers to, **Gapply** performs a sequence of tests.

- If **fn** is the name of a free variable in **v**, then it refers to the value **select_op(free_value(fn,v), ap)** where **free_value(fn,v)** gives the value associated with **fn** in **v** and **select_op** applies the component selector list **ap** to that value; else
- if **fn** is one of the standard identifiers **FALSE** or **TRUE**, it refers to **Gfalse** or **Gtrue**; else
- if **fn** is the name of a standard function, then it refers to that function applied to the actual parameter list **ap**; else
- **fn** refers to a declared function applied to the parameter list **ap**.

(The values of declared scalar constants and other constants declared with a constant declaration are obtained by referring to them as functions of zero arguments.)

5.3 Free Variables

The free variable environment **v** which is used in expression evaluation by **gF(e,c,v,n,x)** is an association list which maps litatom names into marked, typed values. An identifier in an expression is treated as a free-variable iff its name is the key of some pair in **v**. The interpretation of identifiers by **Gapply** gives preference to free variables over function names. Thus, if **'A** appears as the name of a free variable in **v**, then **Gapply** interprets **A(I)** as referring to a component of a composed type rather than as the result of a function application.

An **ENTRY_VALUE** token in an expression also is treated as a free variable.

5.4 Bound Variables

Bound variables are treated directly by the functions **GF_all** and **GF_some** which interpret the logical quantifier operators. (Section 5.5.4.) In Middle Gypsy, these variables are restricted to domains with a finite number of values. These domains are defined by the function **bound_values**.

5.5 Operators

The function **gF** interprets the operators which appear in an expression as applications of the appropriate function as determined by **apply_unary_op** and **apply_binary_op**. For example, **apply_binary_op** interprets **EQ** as an application of the function **Gequal**. The functions which interpret operators, such as **Gequal**, perform the appropriate type checking on their arguments, and then return the value appropriate to that type.

5.5.1 Precedence

Operator precedence in Middle Gypsy (Appendix B) is the same as in Gypsy except that a new precedence level of 4.5 has been introduced for **:>** because Yacc does not allow both left and right associative operators at the same level.

5.5.2 If

An **if_expression** phrase is interpreted directly by **gF**, and **gF** interprets *only* those parts the phrase which are needed to produce a value for the expression.

```
<if_expression, i> :
       IF <expression, b> THEN <expression, p>
       <if_expression_else_part, e>
```

If **gF** interprets **b** above as **Gtrue**, then it interprets **p** and does not interpret **e**. This means that semantic errors are detected in if expressions only when the part of the expression containing the error actually is evaluated by **gF**. Also **gF** does *not* check that **p** gives the same type of value as **e**. The interpretation of the **<if_expression, i>** gives a marked, typed value. When that value is used as an argument to another function, its type is checked by that function.

Checking that all parts of an **if_expression** produce objects of the same type is left to the discretion of language analysis tools. These issue here is not what the semantic error is, but how it is to be detected.

5.5.3 Boolean

The boolean operators are defined as they are in the Gypsy report. The meta-function for each operator evaluates *all* of its arguments, and then performs its operation. If the value of any argument is indeterminate, then so is the result of the boolean operation.

This has an unfortunate effect on the **IMP** operator (which is performed by the **Gimp** function.) If the value of the second argument is indeterminate, then the result of the implication also is indeterminate regardless of the value of the first argument. This becomes significant in cases such as the following. Suppose that an array **a** is declared to have an index range from **0** to **3**, and all elements of this array are **0**. Then under current interpretation, the value of

```
i in [0..3] -> a[i]=0
```

is indeterminate rather than **Gtrue** because **a[i]** is indeterminate for values of **i** outside of the range **[0..3]**.

The problem could be eliminated by defining the boolean operators in terms of the Gypsy if expression. For example, define **x IMP y** to be **IF x THEN y ELSE TRUE**. This kind of definition of the boolean operators could be incorporated directly into the definition of **gF** in the same way that the **if_expression** is done now. The mathematical definition would provide an effective way to analyze the effects of this language change, and this analysis is strongly recommended. (Actually, the conditional interpretation of the boolean operators is the one that was intended in the Gypsy 2.0 report [Good 78].)

5.5.4 Quantifiers

The logical quantifier operators **ALL** and **SOME** are evaluated by the functions **GF_all** and **GF_some**. Middle Gypsy restricts the domain of quantification for these functions to the finite sets which are defined by the function **bound_values**.

5.6 Functions

5.6.1 Standard Functions

The application of a standard function is detected in **Gapply**, and the appropriate function is applied to the actual parameter list **ap**. The names of standard functions begin with the prefix **std_**, and type descriptors are supplied as arguments to the standard functions which produce values associated with the various types -- e.g., for a type descriptor **d** without errors, **std_initial(d)=udv(d)**.

5.6.2 Declared Functions

If an identifier does not refer to a free variable or a standard identifier or a standard function, **Gapply** uses **apply_fun** to interpret it as referring to a declared function. The function **apply_fun** uses the unit referencing function **ref** to get the declaration of the function (or constant) from the library. The formal arguments of the function declaration are checked for semantic errors with the function **farg_check**. This includes using **in_type** to check that the value of an actual parameter is a member of the type of its corresponding formal. If there are no semantic errors, a new free variable environment is created by binding the values of the actual parameters to the names of the corresponding formals.

The mathematical interpretation of a declared function is defined by its **opt_entry_specification** and **opt_exit_specification** phrases, *not* by its **procedure_body** phrase. The **procedure_body** phrase defines its procedural interpretation.

If the value of the entry specification of a declared function is not **Gtrue**, then the value of the function is interpreted as indeterminate. Thus, the entry specification determines a domain of values on which the function *may* be determinate. The function also may be indeterminate for some values in this domain.

The exit specification of a declared function is regarded as an axiom about the function in which the identifier **RESULT** is an abbreviation for the result of applying the function to its arguments. For example, if **DF** is a declared function of **k** arguments, then **RESULT** is an abbreviation for **DF(a1,...,ak)**. If the (normal) exit specification is in the form of a definition, **RESULT = e**, for some expression **e**, then the mathematical value of the function is interpreted as **F(e,c,v,n-1,x)**. (Recall that the value of **F(e,c,v,n,x)** becomes indeterminate when **n=0**.) If the exit specification is not in this definitional form, then the value of the function is indeterminate.

Proof directives are ignored by **gF**. Their interpretation is left to the various analysis tools.

5.7 Conditions

The presence of actual condition parameters in a expression has no effect on the value of **gF** except to produce a semantic error if they violate the conditions stated in the last sentence of the first paragraph of Section 8.5.3 of the Gypsy report. Otherwise, they are completely ignored by **gF**.

5.8 Pre-computable Expressions

A *pre-computable* expression **e** in scope **c** in sentence **x** is one for which **determinate(precomputable_F(e,c,x))=T** where

```
precomputable_F(e,c,x) = F(e,c,empty_map,0,x)
```

Stating this property falls outside of the scope of the definitions in Appendix D.1.

5.9 Lemmas

A Gypsy lemma states a property which is *intended* to be a theorem about the mathematical interpretation of an expression.

A lemma has the form

```
<lemma_declaration, d> :
       LEMMA IDENTIFIER <opt_external_data_objects, a> EQUAL
             <non_validated_specification_expression, b>

<non_validated_specification_expression, se> :
       <expression, e>
     | proof_directive <expression, e>
     | OPEN_PAREN proof_directive <expression, e> CLOSE_PAREN
```

The property which is intended to be a theorem for a lemma declaration **d** which appears in scope **c** in a sentence **x** is

```
free_in_type(a,c,v,x) -> (some n1:integer, F(e,c,v,n1,x) = Gtrue)
```

(where **a** is the **opt_external_data_objects** phrase).

The hypothesis of the intended theorem, **free_in_type(a,c,v,x)**, says that every free variable which appears as a formal argument to the lemma declaration satisfies its type requirement. To be more precise, suppose that formal argument **zi** has **type_specification, ti** in the lemma declaration, and let **tdi=type_desc(ti,c,nil,x)**. Then, the type requirement for this free variable is

```
      tdi = type(mapped_value(v,gname(zi)))
and in_type(tdi),mapped_value(v,gname(zi)))
```

This says that the type descriptor for the free variable is the descriptor constructed from the type specification for the formal argument, and the value associated with the free variable is a member of that type.

The conclusion of the intended theorem **`F(e,c,v,n,x)=Gtrue`** says that there must be some integer **`n1`** sufficiently large so that the mathematical interpretation of the expression **`e`** gives the (determinate, boolean) value **`Gtrue`**.

Proof directives are ignored by **`gF`**. Their interpretation is left to the various analysis tools.

Chapter 6

COMPUTER PROGRAM OPERATION

The function **gP(m,c,s,n,x)** defines the procedural interpretation of Middle Gypsy in a manner similar to the Micro Gypsy interpreter defined in [Young 88]. The function **gP** interprets the sequence of symbols **m** as describing a control mechanism, and it interprets **s** as describing the state upon which that mechanism operates. A state is a mapping which maps names of state components into marked, typed values (and **s** also has some other special components). The symbols in **m** are interpreted as referring to constants, components of the state **s**, component control mechanisms and composite control mechanisms. The function **gP** defines the effect of the mechanism described by **m** by defining the final state which it produces when it begins operating on the initial marked state **s**. An important difference between **gF** and **gP** is that **gF** produces a marked, typed value whereas **gP** produces a marked state.

The parameters and the result of **gP(m,c,s,n,x)** are as follows:

m	is the parse tree which describes a computer program mechanism.
c	is the (litatom) name of the Gypsy scope in which **m** appears.
s	is the marked, initial state on which the mechanism **m** begins to operate.
n	is the maximum number of steps the mechanism is allowed to perform.
x	is the parse tree of the program description sentence which defines the library containing the declarations of the Gypsy units referred to by **m**.
gP(m,c,s,n,x)	is the marked, final state which results from operating the mechanism on the initial state **s** for at most **n** steps. If the mechanism has not completed all of its operation in **n** steps, then the final state is marked as indeterminate.

The function **gP(m,c,s,n,x)** interprets the tree **m** as describing a control mechanism which operates on the state **s**. The definition of **gP** describes the effects of several different primitive mechanisms which can be referred to by **m**, and it describes the effects of various ways of building composite mechanisms from component ones. The function **gP** is defined so that the effects of the primitive mechanisms cannot be changed by any Gypsy unit which may be declared in **x**. The sentence **x** may contain declarations of procedures and executable functions which describe mechanisms which are composed of primitive and non-primitive mechanisms. But the definition of **gP** does not allow these declarations to change the interpretation of the primitive mechanisms.

There is another important difference between **gF** and **gP**. The definition of **gF** defines a *unique* function. The definition of **gP** does not. Instead, it defines a non-empty *set* of functions by defining axioms which members of that set must satisfy. Each function in this set describes the interpretation of a potential *implementation* of Middle Gypsy. The implementation-independent interpretation of a Gypsy sentence is defined by the axioms for **gP**. The implementation-dependent interpretation is defined by a particular function in the set defined by the axioms. The axioms describe the behavior of all mechanisms which can be described by all possible implementations of Middle Gypsy. Each function which satisfies the axioms describes the behavior of all mechanisms which can be described one particular implementation.

The axioms for **gP** impose a set of conventions for signalling Gypsy conditions. The conventions identify conditions which a mechanism is *required* to report, and they impose certain requirements on how that must be done. But they also *allow* a mechanism to signal other implementation-dependent conditions. These conditions can report the exhaustion of finite resources (such as time, space or word size), other kinds of operation failure or whatever other situations might be appropriate to a particular implementation.

The axioms for **gP** are stated by using *constrain* events in the Boyer-Moore logic [Boyer 89]. A constrain event identifies axioms which provide a partial specification of one more functions. To ensure that the axioms are consistent, and hence that there is at least one function that satisfies the axioms, admission of a constrained function to the Boyer-Moore logic requires that a *witness* function which satisfies the axioms be identified. Thus, when a constrained function, or any function which is defined in terms of a constrained function, is admitted to the Boyer-Moore logic, the function admitted represents a non-empty set of functions.

The definition of **gP** is not constrained directly. It is defined in terms of other functions which are constrained. Many of these constraints are stated in terms of functions which are used to define **gF**. This is what establishes an important relation between the mathematical and the procedural interpretations of Middle Gypsy. The mathematical interpretation provides the basis for defining the procedural one.

6.1 Program State

The program state **s** of **gP(m,c,s,n,x)** is a marked object. The function **gP** produces an indeterminate state when **n=0** or when interpreting **PENDING**. The function **gP** also is strict on **s** -- i.e., if **s** is indeterminate, then so is **gP(m,c,s,n,x)**. If **s** is indeterminate, then **gP(m,c,s,n,x)=s**.

The unmarked object of a marked state **s** is a name-value map which is given by the function **map(s)**. This mapping maps from the names of state components into marked, typed values; and it also may have the following special components:

'cond~	The value of the current condition.
'result~	The value of the current expression.
'var	The list of the names of all state components whose values the mechanism **m** is allowed to change. The list is a list of pairs **[n,m]** where **n** is the name of a state component, and **m** is either **'formal** or **'local**.
'const	The list of the names of all state components whose values the mechanism **m** is *not* allowed to change. The list is a list of pairs of the same form as the **'var** list.
'cond	The list of the (litatom) names of all declared conditions. The list is a list of pairs of the same form as the **'var** list.
'keep~	The parse tree of the keep specification for this mechanism.
'entry	The marked, typed value of the **entry** specification interpreted on the initial state.
'exit	The marked, typed value of the **exit** specification interpreted on the final state.
'keep	The marked, typed logical conjunction of all **keep** specifications interpreted.
'assert	The marked, typed logical conjunction of all **assert** specifications interpreted.

(The **~** character is used on these names so that they do not conflict with the names of Gypsy identifiers which may name other components in the state.)

The value of **cond~(s)** controls condition handling, and **result~(s)** is the marked, typed value that results when **gP** interprets expressions. When interpreting a declared procedure or function, the **var(s), const(s)** and **cond+(s)** lists contain the names of all formal and local variable, constant and condition objects which are declared (either explicitly or implicitly). (**Cond+** is used for this function name to avoid conflict with Boyer-Moore **cond**.) The value of **keep~(s)** is the parse tree of the keep specification operating constraint which applies to the mechanism **m**. This parse tree is in the state because it continually needs to be re-evaluated throughout the operation of the mechanism. The values of **entry(s), exit(s), keep(s)** and **assert(s)** are marked, boolean values which indicate whether the mechanism is operating according to its stated constraints.

6.2 Conditions

The value of **cond~(s)** is the current Gypsy condition of state **s**, and this value controls the use of condition handlers during the interpretation of a mechanism **m** by **gP(m,c,s,n,x)**. This value of **cond~(s)** may be **'normal, 'routineerror** or the name of some other condition (e.g., **spaceerror** or one of the conditions named in the list **cond(s)**).

Every **begin_composition, case_composition, if_composition** and **loop_composition** phrase has an **opt_condition_handlers** phrase. The last step in interpreting these compositions is to apply the function **GP_cond** which uses **cond~(s)** to select the appropriate condition handler for interpretation.

6.3 Expressions

Expressions in Gypsy have a dual interpretation. They are interpreted mathematically by **gF** and procedurally by **gP**. The two interpretations are related, but different. The function **gF** interprets the expression as describing a composition of mathematical operations; whereas **gP** interprets the expression as describing a composition of computer program mechanisms. The mathematical interpretation by **gF(e,c,v,n,x)** maps an expression **e** and a *free-variable environment* **v** into a *marked, typed value*. The procedural interpretation by **gP(e,c,s,n,x)** maps an expression **e** and a *marked state* **s** into a *marked state* by using the function **gPF(e,c,s,n,x)**.

The parameters and the result of **gPF(e,c,s,n,x)** are as follows:

e is the parse tree of an expression which describes a mechanism.

c is the (litatom) name of the Gypsy scope in which **e** appears.

s is the marked, initial state on which the expression mechanism **e** begins to operate.

n is the maximum number of steps that the expression mechanism is allowed to operate.

x is the parse tree of the program description sentence which defines the library containing the declarations of the Gypsy units referred to by **e**.

gPF(e,c,s,n,x)
is the marked, final state which results from operating the mechanism on the initial state **s** for at most **n** steps. If the mechanism has not completed all of its operations in **n** steps, then the final state is marked as indeterminate.

The function **gPF** maps indeterminate states into indeterminate states; and for determinate states, the desired relation between **gF** and **gPF** is the following where **gPFs=gPF(e,c,s,n,x)**:

```
     determinate(gPFs)
     and   cond~(gPFs)='normal
 -> (some n1: integer,
           determinate(gF(e,c,map(s),n1,x))
     and result~(gPFs)=gF(e,c,map(s),n1,x))
```

In this relation, the name-value mapping **map(s)** is interpreted by **gF** as a free-variable environment.

This relation says that if **gPFs** is a determinate state with a normal condition, then there must exist some integer **n1** such that the value of **F(e,c,s,n1,x)** is determinate and that **result~(gPFs)** is *that* determinate value. That is, **result~(gPFs)** is the correct mathematical result as defined by **gF**.

This relation *does* allow **gPFs** to be a determinate state with a condition other than **'normal** for a state **s** for which **gF** can produce a determinate value. Thus, even though **gF** can produce a determinate value for **s**, this relation allows **gPF** to report an implementation-dependent, abnormal condition which indicates that the mechanism cannot complete its operation for this particular value of **s**. For example, it may signal an abnormal condition which reports that adequate resources are not available.

This relation, however, *does not* allow **gPFs** to be a determinate state with a normal condition if **gF(e,c,s,n1,x)** is indeterminate for all values of **n1** (because, in this case, the hypotheses of the relation would be true and the conclusion would be false). This implies that if there is no value of **n1** for which **gF** is determinate, then **gPF** *must* report this by signalling some condition other than **'normal**.

In general, it is not possible to define **gF** and **gPF** so that this relation always is satisfied. This is because the expression **e** may refer to declared functions, and they may be declared so that the relation is *not* satisfied. However, **gPF** is defined so that the relation is satisfied provided the declared functions are declared properly. (Strictly speaking, the previous statement is a conjecture. We believe the conjecture to be a theorem, but it has not been proved.)

6.3.1 Constants and Identifiers

Rather than referring to elements of sets as they do when interpreted by **gF**, constants which are referred to by tokens in an expression are interpreted by **gPF** as describing parts of the state of a program mechanism.

Identifiers are interpreted in a similar way by both **gF** and **gPF**. The function **gPF** interprets an identifier as referring to a component of a program state in a manner similar to the way the **gF** treats an identifier as referring to a free variable. Instead of using the function **Gapply** as **gF** does, **gPF** uses a similar function **GP_apply**. If the identifier is the name of a state component, **GP_apply** interpret it as referring to that component of the state; otherwise, **GP_apply** interprets it as referring to a function.

6.3.2 Standard Functions and Operators

To make it *possible* for the relation between **gF** and **gPF** stated in Section 6.3 to be satisfied, **gPF** is defined with a recursive structure similar to **gF**. The recursion of **gF** ends by applying functions which define the mathematical interpretation of the Gypsy operators and standard functions. For example, **gF** interprets the symbol **PLUS** as an application of the function **Gplus(v1,v2)** to the marked, typed values **v1** and **v2**. The function **gPF** has the same recursive structure as **gF**; but instead of applying **Gplus(v1,v2)** to marked, typed values, it applies **GPF_Gplus(sv1,sv2,s0)** to marked states. The states **sv1** and **sv2** contain the operands for the plus operation, and the state **s0** is the state from which the result of the plus operation is composed.

GPF_Gplus(sv1,sv2,s0) first checks the two operand arguments to ensure that both are determinate and have normal conditions. If so, then it applies the implementation-dependent function **p_Gplus** to the **result~** components of the operand states and returns a state which is composed from **s0** by putting the result of the addition into the **result~** component of **s0** provided **p_Gplus** does not produce an implementation error.

```
GPF_Gplus(sv1,sv2,s0) = p_Gplus(result~(sv1), result~(sv2), s0)
```

The implementation-dependent function **p_Gplus(v1,v2,s0)** is a constrained to satisfy following axioms where **sf=p_Gplus(v1,v2,s0)**

```
A1. determinate(sf)
    and   cond~(sf)='normal
 ->                 determinate(Gplus(v1,v2))
    and sf=store_value('result~, Gplus(v1,v2), s0)

A2.     determinate(sf)
    and  not (cond~(sf)='normal)
 ->     implementation_constrained(sf,s0)
    and member(cond~(sf), '(routineerror, spaceerror))
```

The axiom **A1** is similar to the relation between **gF** and **gPF** stated in Section 6.3. It says that if **gf** is a determinate state with a normal condition, then the *mathematical* function **Gplus(v1,v2)** is determinate, and **sf** is equal to **s0** except that the **'result~** component of **sf** has the correct mathematical result **Gplus(v1,v2)**. This *does require* the implementation-dependent function **p_Gplus** to produce the correct mathematical result when it produces a determinate state with a normal conditions, but it *does not allow* **p_Gplus** to produce a determinate state with a normal condition when the mathematical result is indeterminate.

Additionally, the axiom **A2** says that if **gf** is an determinate state with an abnormal condition, then **sf** is equal to **s0** except that the **'cond~** components may be different, and the **cond~** component of **sf** must be either **'routineerror** or **'spaceerror**. This *allows* the implementation-dependent function **p_Gplus** to signal **'routineerror** and **'spaceerror** at its discretion if, for some reason, the operation cannot be performed.

Each of the functions which is applied by **gPF** to interpret the operators and standard functions is treated in a similar way. This same approach also is used for literal constants which may appear in a expression (e.g., a **DIGIT_LIST** token) because the mechanism being described by **gPF** also may fail for certain values of these constants.

6.3.3 Declared Functions

The function **gPF** uses **GPF_apply_fun** to interpret declared functions in a way analogous to the way that **gF** uses **apply_fun**. The definitions of these two functions are quite similar, but there are some important differences which define the essential differences between the mathematical interpretation of a Gypsy function declaration by **gF** and its procedural interpretation by **gP**. For the mathematical interpretation, the function **apply_fun** produces a marked, typed value from the **opt_entry_specification** and **opt_exit_specification** parts of the function declaration. For the procedural interpretation, the function **GFP_apply_fun** produces a marked state from the **procedure_body** of the declaration.

In the mathematical interpretation, the entry and exit specifications of the function declaration are interpreted as defining a *relation* between the members of a domain set and a range set of a *mathematical* *function*. If this relation is an equality, it is interpreted as the definition of the function. The *only* parts of a **procedure_body** which effect the mathematical interpretation of a function declaration are the entry and exit specification. The other parts of the body are ignored.

The procedural interpretation of a function declaration interprets the **procedure_body** as describing the *operation* of a *mechanism* on a program state. This interpretation is the same as that of the **procedure_body** of a procedure declaration. The only small, but important, difference is that the program state for a function declaration has a special component named **'result** whose value is interpreted as the value of the function. The procedural interpretation of a **procedure_body** interprets the entry and exit specifications as operating constraints which are *intended* to be satisfied by the mechanism described by the **procedure_body**.

If we let **e=DF(a1,...,an)** where **DF** is the name of a declared function, then the desired relation between its mathematical and its procedural interpretation is the one stated in Section 6.3. If the procedural interpretation produces a determinate state with a normal condition, then the mathematical interpretation must be determinate, and the result component of the state must be the correct mathematical result as defined by **gF**.

The function **DF** certainly can be declared so that this relation is *not* satisfied. But (and this is another unproved conjecture) if the mechanism begins operating on an initial state **s** and produces a determinate state with a normal condition on which the exit constraint is satisfied, then the relation *is* satisfied (provided the mathematical interpretation **gF(e,c,map(s),n1,x))** is determinate for some **n1**).

6.4 Primitive Mechanisms

The function **gP(m,c,s,n,x)** may interpret the symbols in **m** as describing any one of several primitive mechanisms which operate on the program state **s**. The primitive mechanisms are signal, leave, data assignment, new, remove, move, procedure call and local var, const and cond. Each of these mechanisms (except for signal and leave) are constrained in a manner similar to the operators and standard functions so that implementations of these primitive mechanisms are allowed to signal implementation-dependent conditions.

The effect of the signal and leave mechanisms is simply to set the value of **cond~(s)**. The leave mechanism sets it to **'leave**, and the loop composition uses this to terminate iteration.

The effect of the data assignment, new and remove mechanisms is to change the value of a state component named on the **'var** list. The **assignment_statement**, **new_statement** and **remove_statement** phrases that describe these mechanisms each have a singe **name_expression**. The **IDENTIFIER** token at the beginning of the **name_expression** identifies the name of the state component whose value is changed. The remainder of the name expression identifies which particular component of a structured value is changed.

Depending on its form, the move mechanism is interpreted as producing the same effect as either new followed by remove or data assignment followed by remove, except that the move operation is done in one step rather than two.

Calls of declared procedures are handled in a manner very similar to the treatment of declared executable functions. The unit referencing function **ref** is used to get the declaration of the procedure from the library. A program state is created in which the formal parameters are bound to the corresponding actuals. Type compatibility and aliasing constraints on the actual parameters are checked. A state component is created for the entry value of each formal parameter.

The body of the procedure is then interpreted within this new environment. The declaration of local var, const and cond objects modify the state by extending the **'var, 'const** and **'cond** list in the state, and they allocate new, initialized state components for local var and const objects. After interpreting the body of the procedure, the values of the actual parameters passed as var parameter slots are copied back to the calling environment. Because of the various constraints on the Gypsy procedure call mechanism, this *call by value-result* parameter passing scheme is equivalent for Gypsy to *call by reference* in the absence of indeterminacy. In particular, the *only* potential effects of a Gypsy procedure call are changes to the var parameters and return of a non-normal condition.

6.5 Composite Mechanisms

Composite mechanisms are described by Gypsy procedure declarations; and as discussed in Section 6.3.3, they also may be described by function declarations. The composition are sequential (**;**), if, case and loop. These are all defined directly in **gP**.

6.6 Operating Constraints

An operating constraint states a property which (the conceptual model of) the operation of a program mechanism is *intended* to have. The procedural interpretation of a mechanism by **gP(m,c,s,n,x)** uses the **'entry, 'exit, 'keep** and **'assert** components of the program state to indicate whether the operation of **m** does satisfies its intended operating constraints.

The procedural interpretation of entry and exit specifications is that their expressions are evaluated *once* upon entering and exiting a **procedure_body** phrase, and the values of these expressions become the values which are associated with the **'entry** and **'exit** components of the state **s**. These evaluations are performed without the local var and const components in the state, and the expression evaluated for a case exit is the one that matches the current condition **cond~(s)**.

The values of the **'keep** and **'assert** components of the state begin a value of **Gtrue**. Every time a keep or assert specification is encountered in the procedural interpretation, the specification is evaluated; and its value conjoined (with **Gand**) to the current value of the **'keep** or **'assert** component of the state.

This procedural interpretation of entry, exit, keep and assert evaluates their expressions with the *mathematical* interpreter **gF(e,c,s,v,x)** with the current state **s** defining the free-variable environment because the operating constraints are *mathematical* constraints on the operation of the *mechanism*. The one exception to this is the run-time validation of assert specifications.

Proof directives are ignored by **gP**. Their interpretation is left to the various analysis tools.

6.7 Run-Time Validation

The run-time validation of an assert specification

```
ASSERT e OTHERWISE c
```

is interpreted procedurally as the if statement

```
IF not e THEN SIGNAL c
```

and the value of **`e`** is conjoined to the **`'assert`** component of the state as usual. However, because **`e`** now appears as the boolean test of an if statement, it is evaluated as a program expression with **`gPF`** rather than using the mathematical interpretation **`gF`**.

Appendix A
LALR Grammar

The ASCII character strings in this section are defined in the regular expression notation of [Aho 86] -- **`*`** means zero or more repetitions, **`|`** means or, and **`empty`** is the string with no characters.

A.1 White Space

White space is a (possibly empty) sequence of Gypsy comments and ASCII space, tab, carriage return, line feed characters.

A Gypsy comment is a sequence of ASCII characters which begins with the character **`{`** and ends with the first subsequent **`}`** character.

```
comment : { comment_character* }
```

where **`comment_character`** is any ASCII character except **`}`**.

A.2 Tokens

The lexemes of Gypsy tokens may be separated by white space; and unless stated otherwise, a lexeme may not contain embedded white space. (The admission of white space between lexemes is not covered by the mathematical definition.) The lexemes of the tokens of the grammar are listed below.

Reserved Words — are recognized by **`reserved_wordp(x)`**. Each of the following reserved words are tokens:

```
ADJOIN, ALL, AND, APPEND, ARRAY, ASSERT, ASSUME, AWAIT, BEFORE,
BEGIN, BEHIND, BINARY, BLOCK, BUFFER, CASE, CBLOCK, CENTRY,
CEXIT, COBEGIN, COND, CONST, DECIMAL, DIFFERENCE, DIV, EACH,
ELEMENT, ELIF, ELSE, END, ENTRY, EQ, EXIT, EXTENDS, FI, FROM,
FUNCTION, GE, GIVE, GT, HEX, HOLD, IF, IFF, IMP, INPUT, IN,
INTO, INITIALLY, INTERSECT, IS, KEEP, LE, LEAVE, LEMMA, LOOP,
LT, MAPOMIT, MAPPING, MOD, MOVE, NAME, NE, NEW, NORMAL, NOT,
OCTAL, OF, OMIT, ON, OR, OTHERWISE, OUTPUT, PENDING, PROCEDURE,
PROVE, RECEIVE, RECORD, REMOVE, SCOPE, SEND, SEQ, SEQOMIT,
SEQUENCE, SET, SIGNAL, SOME, SUB, THEN, TO, TYPE, UNION,
UNLESS, VAR, WHEN, WITH,

ALIAS, EXPORT, IMPORT, MULTIPLECOND, NONE, SPACE, STRING,
VALUE.
```

For compatibility with Gypsy, this list includes all of the Gypsy reserved words.

The lexeme for the token is any sequence of letters which matches the word. The letters may be any combination of upper and lower case. For example, the lexeme for token **`END`** is **`(E | e) (N | n) (D  | d)`**. Some of the reserved words have alternate lexemes which are composed of special symbols. They are are marked with **`[#]`** below.

The following sequences of special symbols are lexemes for tokens. They are recognized by **`special_symbolp(x)`**. The ones marked **`[#]`** are alternate lexemes for reserved words.

AND	**&** [#]
APPEND	**@** [#]

CLOSE_PAREN	**] |)**
COLON	**:**
COLON_EQUAL	**:=**
COLON_GT	**:>**
COMMA	**,**
DOT	**.**
DOT_DOT	**..**
EQ	**=** [#]
LT	**<** [#]
LT_COLON	**<:**
GT	**>** [#]
IMP	**->** [#]
MINUS	**-**
OPEN_PAREN	**[| (**
PLUS	**+**
SEMI_COLON	**;**
STAR	*****
STAR_STAR	******

The following are tokens:

CHARACTER_VALUE
is recognized by **character_valuep(x)**. The lexeme of a **CHARACTER_VALUE** token is a single printable ASCII character embedded in single quote marks.

```
CHARACTER_VALUE : ' printable_char '
```

where **printable_char** is any **letter**, **digit** or ASCII character with code 32-47, 58-64, 91-96 or 123-126.

DIGIT_LIST is recognized by **digit_listp(x)**.

```
DIGIT_LIST : digit hexdigit*

digit : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

hexdigit : digit | A | B | C | D | E | F
                 | a | b | c | d | e | f
```

ENTRY_VALUE is recognized by **entry_valuep(x)**.

```
ENTRY_VALUE : IDENTIFIER '
```

IDENTIFIER is recognized by **identifierp(x)**. The lexeme of an **IDENTIFIER** token may be any of the following except that it may *not* be the lexeme of any of the reserved words.

```
IDENTIFIER : letter idtail*

idtail : opt_score letter_or_digit

opt_score : _ | empty

letter_or_digit : letter | digit

letter :
         A | B | C | D | E | F | G | H | I | J | K | L | M
       | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
       | a | b | c | d | e | f | g | h | i | j | k | l | m
       | n | o | p | q | r | s | t | u | v | w | x | y | z
```

STRING_VALUE is recognized by **string_valuep(x)**. Upper and lower case are *not* interchangeable in the lexeme of a **STRING_VALUE** token, and the lexeme *may* contain embedded white space.

```
STRING_VALUE : " ( non_quote | "" )* } "
```

where **non_quote** is any ASCII character except the double quote mark **"**.

The remainder of this section is the input file which defines the Gypsy tokens for the lexical analyzer program Lex. The Lex input formats are summarized in [Aho 86].

```
%e 1500
%k 2000
%p 5500
%a 3000

digit [0-9]
hexdigit {digit}|[a-fA-F]
letter [a-zA-Z]
char32-47 [\ \!\"\#\$\%\&\’\(\)\*\+\,\-\.\/]
char58-64 [\:\;\<\=\>\?\@]
char91-6 [\[\\\]\^\_\‘]
char123-6 [\{\|\}\~]
printable_char {letter}|{digit}|{char32-47}|{char58-64}|{char91-6}|{char123-6}
comment \{[^\}]*\}
white [ \t\n\\]|{comment}
%%
[aA][dD][jJ][oO][iI][nN]  {
                printf ("ADJOIN - %s\n", yytext);
                return(ADJOIN);
                }
[aA][lL][lL]  {
                printf ("ALL - %s\n", yytext);
                return(ALL);
                }
[aA][nN][dD]  {
                printf ("AND - %s\n", yytext);
                return(AND);
                }
\&  {
                printf ("AND - %s\n", yytext);
                return(AND);
                }
[aA][pP][pP][eE][nN][dD]  {
                printf ("APPEND - %s\n", yytext);
                return(APPEND);
                }
\@  {
                printf ("APPEND - %s\n", yytext);
                return(APPEND);
                }
[aA][rR][rR][aA][yY]  {
                printf ("ARRAY - %s\n", yytext);
                return(ARRAY);
```

```
                }
[aA][sS][sS][eE][rR][tT]  {
                printf ("ASSERT - %s\n", yytext);
                return(ASSERT);
                }
[aA][sS][sS][uU][mM][eE]  {
                printf ("ASSUME - %s\n", yytext);
                return(ASSUME);
                }
[aA][wW][aA][iI][tT]  {
                printf ("AWAIT - %s\n", yytext);
                return(AWAIT);
                }
[bB][eE][fF][oO][rR][eE]  {
                printf ("BEFORE - %s\n", yytext);
                return(BEFORE);
                }
[bB][eE][gG][iI][nN]  {
                printf ("BEGIN - %s\n", yytext);
                return(BEGIN);
                }
[bB][eE][hH][iI][nN][dD]  {
                printf ("BEHIND - %s\n", yytext);
                return(BEHIND);
                }
[bB][iI][nN][aA][rR][yY]  {
                printf ("BINARY - %s\n", yytext);
                return(BINARY);
                }
[bB][lL][oO][cC][kK]  {
                printf ("BLOCK - %s\n", yytext);
                return(BLOCK);
                }
[bB][uU][fF][fF][eE][rR]  {
                printf ("BUFFER - %s\n", yytext);
                return(BUFFER);
                }
[cC][aA][sS][eE]  {
                printf ("CASE - %s\n", yytext);
                return(CASE);
                }
[cC][bB][lL][oO][cC][kK]  {
                printf ("CBLOCK - %s\n", yytext);
                return(CBLOCK);
                }
[cC][eE][nN][tT][rR][yY]  {
                printf ("CENTRY - %s\n", yytext);
                return(CENTRY);
                }
[cC][eE][xX][iI][tT]  {
                printf ("CEXIT - %s\n", yytext);
                return(CEXIT);
                }
[cC][oO][bB][eE][gG][iI][nN]  {
                printf ("COBEGIN - %s\n", yytext);
                return(COBEGIN);
                }
[cC][oO][nN][dD]  {
                printf ("COND - %s\n", yytext);
                return(COND);
                }
[cC][oO][nN][sS][tT]  {
                printf ("CONST - %s\n", yytext);
                return(CONST);
                }
[dD][eE][cC][iI][mM][aA][lL]  {
                printf ("DECIMAL - %s\n", yytext);
                return(DECIMAL);
                }
```

```
[dD][iI][fF][fF][eE][rR][eE][nN][cC][eE]  {
                printf ("DIFFERENCE - %s\n", yytext);
                return(DIFFERENCE);
                }
[dD][iI][vV]  {
                printf ("DIV - %s\n", yytext);
                return(DIV);
                }
[eE][aA][cC][hH]  {
                printf ("EACH - %s\n", yytext);
                return(EACH);
                }
[eE][lL][eE][mM][eE][nN][tT]  {
                printf ("ELEMENT - %s\n", yytext);
                return(ELEMENT);
                }
[eE][lL][iI][fF]  {
                printf ("ELIF - %s\n", yytext);
                return(ELIF);
                }
[eE][lL][sS][eE]  {
                printf ("ELSE - %s\n", yytext);
                return(ELSE);
                }
[eE][nN][dD]  {
                printf ("END - %s\n", yytext);
                return(END);
                }
[eE][nN][tT][rR][yY]  {
                printf ("ENTRY - %s\n", yytext);
                return(ENTRY);
                }
[eE][qQ]  {
                printf ("EQ - %s\n", yytext);
                return(EQ);
                }
\=  {
                printf ("EQUAL - %s\n", yytext);
                return(EQUAL);
                }
[eE][xX][iI][tT]  {
                printf ("EXIT - %s\n", yytext);
                return(EXIT);
                }
[eE][xX][tT][eE][nN][dD][sS]  {
                printf ("EXTENDS - %s\n", yytext);
                return(EXTENDS);
                }
[fF][iI]  {
                printf ("FI - %s\n", yytext);
                return(FI);
                }
[fF][rR][oO][mM]  {
                printf ("FROM - %s\n", yytext);
                return(FROM);
                }
[fF][uU][nN][cC][tT][iI][oO][nN]  {
                printf ("FUNCTION - %s\n", yytext);
                return(FUNCTION);
                }
[gG][eE]  {
                printf ("GE - %s\n", yytext);
                return(GE);
                }
[gG][iI][vV][eE]  {
                printf ("GIVE - %s\n", yytext);
                return(GIVE);
                }
[gG][tT]  {
```

```
                printf ("GT - %s\n", yytext);
                return(GT);
                }
\>  {
                printf ("GT - %s\n", yytext);
                return(GT);
                }
[hH][eE][xX]  {
                printf ("HEX - %s\n", yytext);
                return(HEX);
                }
[hH][oO][lL][dD]  {
                printf ("HOLD - %s\n", yytext);
                return(HOLD);
                }
[iI][fF]  {
                printf ("IF - %s\n", yytext);
                return(IF);
                }
[iI][fF][fF]  {
                printf ("IFF - %s\n", yytext);
                return(IFF);
                }
[iI][mM][pP]  {
                printf ("IMP - %s\n", yytext);
                return(IMP);
                }
\-\>  {
                printf ("IMP - %s\n", yytext);
                return(IMP);
                }
[iI][nN][pP][uU][tT]  {
                printf ("INPUT - %s\n", yytext);
                return(INPUT);
                }
[iI][nN]  {
                printf ("IN - %s\n", yytext);
                return(IN);
                }
[iI][nN][tT][oO]  {
                printf ("INTO - %s\n", yytext);
                return(INTO);
                }
[iI][nN][iI][tT][iI][aA][lL][lL][yY]  {
                printf ("INITIALLY - %s\n", yytext);
                return(INITIALLY);
                }
[iI][nN][tT][eE][rR][sS][eE][cC][tT]  {
                printf ("INTERSECT - %s\n", yytext);
                return(INTERSECT);
                }
[iI][sS]  {
                printf ("IS - %s\n", yytext);
                return(IS);
                }
[kK][eE][eE][pP]  {
                printf ("KEEP - %s\n", yytext);
                return(KEEP);
                }
[lL][eE]  {
                printf ("LE - %s\n", yytext);
                return(LE);
                }
[lL][eE][aA][vV][eE]  {
                printf ("LEAVE - %s\n", yytext);
                return(LEAVE);
                }
[lL][eE][mM][mM][aA]  {
                printf ("LEMMA - %s\n", yytext);
```

```
                return(LEMMA);
                }
[lL][oO][oO][pP]  {
                printf ("LOOP - %s\n", yytext);
                return(LOOP);
                }
[lL][tT]  {
                printf ("LT - %s\n", yytext);
                return(LT);
                }
\<  {
                printf ("LT - %s\n", yytext);
                return(LT);
                }
[mM][aA][pP][oO][mM][iI][tT]  {
                printf ("MAPOMIT - %s\n", yytext);
                return(MAPOMIT);
                }
[mM][aA][pP][pP][iI][nN][gG]  {
                printf ("MAPPING - %s\n", yytext);
                return(MAPPING);
                }
[mM][oO][dD]  {
                printf ("MOD - %s\n", yytext);
                return(MOD);
                }
[mM][oO][vV][eE]  {
                printf ("MOVE - %s\n", yytext);
                return(MOVE);
                }
[nN][aA][mM][eE]  {
                printf ("NAME - %s\n", yytext);
                return(NAME);
                }
[nN][eE]  {
                printf ("NE - %s\n", yytext);
                return(NE);
                }
[nN][eE][wW]   {
                printf ("NEW - %s\n", yytext);
                return(NEW);
                }
[nN][oO][rR][mM][aA][lL]  {
                printf ("NORMAL - %s\n", yytext);
                return(NORMAL);
                }
[nN][oO][tT]  {
                printf ("NOT - %s\n", yytext);
                return(NOT);
                }
[oO][cC][tT][aA][lL]  {
                printf ("OCTAL - %s\n", yytext);
                return(OCTAL);
                }
[oO][fF]  {
                printf ("OF - %s\n", yytext);
                return(OF);
                }
[oO][mM][iI][tT]  {
                printf ("OMIT - %s\n", yytext);
                return(OMIT);
                }
[oO][nN]  {
                printf ("ON - %s\n", yytext);
                return(ON);
                }
[oO][rR]  {
                printf ("OR - %s\n", yytext);
                return(OR);
```

```
                }
[oO][tT][hH][eE][rR][wW][iI][sS][eE]  {
                printf ("OTHERWISE - %s\n", yytext);
                return(OTHERWISE);
                }
[oO][uU][tT][pP][uU][tT]  {
                printf ("OUTPUT - %s\n", yytext);
                return(OUTPUT);
                }
[pP][eE][nN][dD][iI][nN][gG]   {
                printf ("PENDING - %s\n", yytext);
                return(PENDING);
                }
[pP][rR][oO][cC][eE][dD][uU][rR][eE]  {
                printf ("PROCEDURE - %s\n", yytext);
                return(PROCEDURE);
                }
[pP][rR][oO][vV][eE]  {
                printf ("PROVE - %s\n", yytext);
                return(PROVE);
                }
[rR][eE][cC][eE][iI][vV][eE]  {
                printf ("RECEIVE - %s\n", yytext);
                return(RECEIVE);
                }
[rR][eE][cC][oO][rR][dD]  {
                printf ("RECORD - %s\n", yytext);
                return(RECORD);
                }
[rR][eE][mM][oO][vV][eE]  {
                printf ("REMOVE - %s\n", yytext);
                return(REMOVE);
                }
[sS][cC][oO][pP][eE]  {
                printf ("SCOPE - %s\n", yytext);
                return(SCOPE);
                }
[sS][eE][nN][dD]  {
                printf ("SEND - %s\n", yytext);
                return(SEND);
                }
[sS][eE][qQ]  {
                printf ("SEQ - %s\n", yytext);
                return(SEQ);
                }
[sS][eE][qQ][oO][mM][iI][tT]  {
                printf ("SEQOMIT - %s\n", yytext);
                return(SEQOMIT);
                }
[sS][eE][qQ][uU][eE][nN][cC][eE]  {
                printf ("SEQUENCE - %s\n", yytext);
                return(SEQUENCE);
                }
[sS][eE][tT]  {
                printf ("SET - %s\n", yytext);
                return(SET);
                }
[sS][iI][gG][nN][aA][lL]  {
                printf ("SIGNAL - %s\n", yytext);
                return(SIGNAL);
                }
[sS][oO][mM][eE]  {
                printf ("SOME - %s\n", yytext);
                return(SOME);
                }
[sS][uU][bB]  {
                printf ("SUB - %s\n", yytext);
                return(SUB);
                }
```

```
[tT][hH][eE][nN]  {
                printf ("THEN - %s\n", yytext);
                return(THEN);
                }
[tT][oO]  {
                printf ("TO - %s\n", yytext);
                return(TO);
                }
[tT][yY][pP][eE]  {
                printf ("TYPE - %s\n", yytext);
                return(TYPE);
                }
[uU][nN][iI][oO][nN]  {
                printf ("UNION - %s\n", yytext);
                return(UNION);
                }
[uU][nN][lL][eE][sS][sS]  {
                printf ("UNLESS - %s\n", yytext);
                return(UNLESS);
                }
[vV][aA][rR]  {
                printf ("VAR - %s\n", yytext);
                return(VAR);
                }
[wW][hH][eE][nN]  {
                printf ("WHEN - %s\n", yytext);
                return(WHEN);
                }
[wW][iI][tT][hH]  {
                printf ("WITH - %s\n", yytext);
                return(WITH);
                }
[aA][lL][iI][aA][sS]  {
                printf ("ALIAS - %s\n", yytext);
                return(ALIAS);
                }
[eE][xX][pP][oO][rR][tT]  {
                printf ("EXPORT - %s\n", yytext);
                return(EXPORT);
                }
[iI][mM][pP][oO][rR][tT]  {
                printf ("IMPORT - %s\n", yytext);
                return(IMPORT);
                }
[mM][uU][lL][tT][iI][pP][lL][eE][cC][oO][nN][dD]  {
                printf ("MULTIPLECOND - %s\n", yytext);
                return(MULTIPLECOND);
                }
[nN][oO][nN][eE]  {
                printf ("NONE - %s\n", yytext);
                return(NONE);
                }
[sS][pP][aA][cC][eE]  {
                printf ("SPACE - %s\n", yytext);
                return(SPACE);
                }
[sS][tT][rR][iI][nN][gG]   {
                printf ("STRING - %s\n", yytext);
                return(STRING);
                }
[vV][aA][lL][uU][eE]  {
                printf ("VALUE - %s\n", yytext);
                return(VALUE);
                }
\]  {
                printf ("CLOSE_PAREN - %s\n", yytext);
                return(CLOSE_PAREN);
                }
\)  {
```

```
                printf ("CLOSE_PAREN - %s\n", yytext);
                return(CLOSE_PAREN);
                }
\:  {
                printf ("COLON - %s\n", yytext);
                return(COLON);
                }
\:\=  {
                printf ("COLON_EQUAL - %s\n", yytext);
                return(COLON_EQUAL);
                }
\:\>  {
                printf ("COLON_GT - %s\n", yytext);
                return(COLON_GT);
                }
\,  {
                printf ("COMMA - %s\n", yytext);
                return(COMMA);
                }
\.  {
                printf ("DOT - %s\n", yytext);
                return(DOT);
                }
\.\.  {
                printf ("DOT_DOT - %s\n", yytext);
                return(DOT_DOT);
                }
\<\:  {
                printf ("LT_COLON - %s\n", yytext);
                return(LT_COLON);
                }
\-  {
                printf ("MINUS - %s\n", yytext);
                return(MINUS);
                }
\[  {
                printf ("OPEN_PAREN - %s\n", yytext);
                return(OPEN_PAREN);
                }
\(  {
                printf ("OPEN_PAREN - %s\n", yytext);
                return(OPEN_PAREN);
                }
\+  {
                printf ("PLUS - %s\n", yytext);
                return(PLUS);
                }
\;  {
                printf ("SEMI_COLON - %s\n", yytext);
                return(SEMI_COLON);
                }
\/  {
                printf ("SLASH - %s\n", yytext);
                return(SLASH);
                }
\*  {
                printf ("STAR - %s\n", yytext);
                return(STAR);
                }
\*\*  {
                printf ("STAR_STAR - %s\n", yytext);
                return(STAR_STAR);
                }
{letter}(_?({letter}|{digit}))*  {
                printf ("IDENTIFIER - %s\n", yytext);
                return(IDENTIFIER);
                }
{digit}({hexdigit})*  {
                printf ("DIGIT_LIST - %s\n", yytext);
```

```
                return(DIGIT_LIST);
                }
\'({printable_char})\'  {
                printf ("CHARACTER_VALUE - %s\n", yytext);
                return(CHARACTER_VALUE);
                }
\"([^\"]|\"\")*\"  {
                printf ("STRING_VALUE - %s\n", yytext);
                return(STRING_VALUE);
                }
{letter}(_?({letter}|{digit}))*\'  {
                printf ("ENTRY_VALUE - %s\n", yytext);
                return(ENTRY_VALUE);
                }
({white})+ ;

.  {
                printf ("UNKNOWN_TOKEN - %s\n", yytext);
                return(UNKNOWN_TOKEN);
                }
```

A.3 Productions

This section contains the input file which defines the production rules of the Middle Gypsy 2.05 grammar for the parser generator program Yacc. The Yacc input formats are summarized in [Aho 86]. The production rules listed here were constructed mechanically from the definitions of the meta-functions in Appendix D.

```
%token ADJOIN ALL AND APPEND ARRAY ASSERT ASSUME AWAIT BEFORE BEGIN BEHIND
%token BINARY BLOCK BUFFER CASE CBLOCK CENTRY CEXIT COBEGIN COND CONST
%token DECIMAL DIFFERENCE DIV EACH ELEMENT ELIF ELSE END ENTRY EQ EXIT
%token EXTENDS FI FROM FUNCTION GE GIVE GT HEX HOLD IF IFF IMP INPUT IN INTO
%token INITIALLY INTERSECT IS KEEP LE LEAVE LEMMA LOOP LT MAPOMIT MAPPING MOD
%token MOVE NAME NE NEW  NORMAL NOT OCTAL OF OMIT ON OR OTHERWISE OUTPUT
%token PENDING PROCEDURE PROVE RECEIVE RECORD REMOVE SCOPE SEND SEQ SEQOMIT
%token SEQUENCE SET SIGNAL SOME SUB THEN TO TYPE UNION UNLESS VAR WHEN WITH
%token ALIAS EXPORT IMPORT MULTIPLECOND NONE SPACE STRING VALUE


%token CLOSE_PAREN COLON COLON_EQUAL COLON_GT COMMA DOT
/*      ) or ]      :     :=          :>       ,     .               */

%token DOT_DOT EQUAL LT_COLON
/*      ..      =     <:                                             */

%token MINUS OPEN_PAREN PLUS SEMI_COLON SLASH STAR STAR_STAR
/*      -     ( or [     +    ;          /     *    **               */


/*   Token for  & is AND.          */
/*   Token for  @ is APPEND.       */
/*   Token for  > is GT.           */
/*   Token for -> is IMP.          */
/*   Token for  < is LT.           */


/*  Balance () [].                                                   */



%token DIGIT_LIST IDENTIFIER CHARACTER_VALUE STRING_VALUE ENTRY_VALUE

%token UNKNOWN_TOKEN

%left ALL SOME   /* Previously QUANTIFIER */
%left IMP IFF
%left OR
%left AND
%left NOT
%left EQUAL EQ NE LT LE GT GE IN SUB
%left APPEND UNION INTERSECT DIFFERENCE
%left ADJOIN OMIT
%right COLON_GT
%left PLUS MINUS LT_COLON    /* binary -  */
%left STAR SLASH DIV MOD
%left UNARY_MINUS  /* unary -  */
%left STAR_STAR

%start program_description

%%

access_specification :
       VAR
     | CONST
       ;

actual_condition_parameters :
```

```
       UNLESS OPEN_PAREN opt_group_name identifier_list CLOSE_PAREN
       ;

arg_list :
       OPEN_PAREN value_list CLOSE_PAREN
       ;

array_type :
       ARRAY OPEN_PAREN type_specification CLOSE_PAREN
          OF type_specification
       ;

assert_specification :
       ASSERT specification_expression
       ;

assignment_statement :
       name_expression COLON_EQUAL expression
       ;

base :
       BINARY
     | OCTAL
     | DECIMAL
     | HEX
       ;

begin_composition :
       BEGIN
       opt_internal_statements
       opt_condition_handlers
       END
       ;

bound_expression :
       identifier_list COLON type_specification COMMA expression   %prec ALL
       ;

case_composition :
       CASE expression
       case_composition_body
       opt_condition_handlers
       END
       ;

case_composition_body :
       empty
     | ELSE             COLON opt_internal_statements
     | IS case_labels COLON opt_internal_statements
       case_composition_body
     ;

case_exit :
       IS case_exit_labels COLON
          non_validated_specification_expression
       ;

case_exit_body :
       case_exit
     | case_exit_body SEMI_COLON case_exit
       ;

case_exit_labels :
       exit_label
     | case_exit_labels COMMA exit_label
       ;

case_labels :
       pre_computable_label_expression
```

```
     | case_labels COMMA pre_computable_label_expression
       ;

component_alterations :
       opt_each_clause  component_assignment
     | opt_each_clause  component_creation
     | opt_each_clause  component_deletion
       ;

component_alterations_list :
       component_alterations
     | component_alterations_list SEMI_COLON component_alterations
       ;

component_assignment :
               selector_list COLON_EQUAL expression
       ;

component_creation :
       BEFORE  selector_list COLON_EQUAL expression
       BEHIND  selector_list COLON_EQUAL expression
       INTO    selector_list COLON_EQUAL expression
       ;

component_deletion :
       SEQOMIT selector_list
     | MAPOMIT selector_list
       ;

component_destination :
       new_dynamic_variable_component
     | TO name_expression
       ;

component_selectors :
       DOT IDENTIFIER
     | arg_list
       ;

conditional_exit_specification :
       CASE OPEN_PAREN case_exit_body CLOSE_PAREN
       ;

constant_body :
       PENDING
     | expression
       ;

constant_declaration :
       CONST IDENTIFIER COLON type_specification COLON_EQUAL constant_body
       ;

element_list :
       value_list
     | range_limits
       ;

empty :  ;

exit_label :
       IDENTIFIER
     | NORMAL
       ;

expression :
       modified_primary_value
     | ALL bound_expression
     | SOME bound_expression
         /* integer/rational unary operator */
```

```
     | MINUS expression  %prec UNARY_MINUS
         /* boolean unary operator */
     | NOT expression
         /* simple relational operator */
     | expression EQ expression
     | expression EQUAL expression  /* Use EQ in parse tree. */
     | expression NE expression
     | expression LT expression
     | expression LE expression
     | expression GT expression
     | expression GE expression
         /* boolean operator */
     | expression AND expression
     | expression OR expression
     | expression IMP expression
     | expression IFF expression
         /* integer/rational operator */
     | expression STAR_STAR expression
     | expression STAR expression
     | expression SLASH expression
     | expression DIV expression
     | expression MOD expression
     | expression PLUS expression
     | expression MINUS expression
         /* seq/set/map operator */
     | expression IN expression
     | expression ADJOIN expression
     | expression OMIT expression
     | expression SUB expression
         /* set/map operator */
     | expression UNION expression
     | expression INTERSECT expression
     | expression DIFFERENCE expression
         /* sequence operator */
     | expression COLON_GT expression
     | expression LT_COLON expression
     | expression APPEND expression
       ;

external_data_objects :
       OPEN_PAREN external_data_objects_list CLOSE_PAREN
       ;

external_data_objects_list :
       similar_formal_data_parameters
     | external_data_objects_list SEMI_COLON similar_formal_data_parameters
       ;

external_operational_specification :
       opt_entry_specification
       opt_exit_specification
       ;

fields :
       similar_fields
     | fields SEMI_COLON similar_fields
       ;

function_declaration :
       FUNCTION IDENTIFIER
       opt_external_data_objects COLON type_specification
       opt_external_conditions EQUAL
       procedure_body
       ;

handler :
       IS identifier_list COLON opt_internal_statements
       ;
```

```
handler_list :
       handler
     | handler_list handler
       ;

identifier_list :
       IDENTIFIER
     | identifier_list COMMA IDENTIFIER
       ;

if_composition :
       IF expression THEN opt_internal_statements
       if_composition_else_part
       opt_condition_handlers
       END
       ;

if_composition_else_part :
       empty
     | ELSE opt_internal_statements
     | ELIF expression THEN opt_internal_statements
       if_composition_else_part
       ;

if_expression :
       IF expression THEN expression
       if_expression_else_part
       ;

if_expression_else_part :
       ELSE expression FI
     | ELIF expression THEN expression
       if_expression_else_part
       ;

internal_data_or_condition_objects :
       access_specification identifier_list COLON type_specification
                                   opt_internal_initial_value SEMI_COLON
     | COND identifier_list SEMI_COLON
       ;

internal_environment :
       internal_data_or_condition_objects
     | internal_environment internal_data_or_condition_objects
       ;

lemma_declaration :
       LEMMA IDENTIFIER opt_external_data_objects EQUAL
             non_validated_specification_expression
       ;

leave_statement :
       LEAVE
       ;

literal_value :
       CHARACTER_VALUE
     | number
     | STRING_VALUE
       ;

local_aliases :
       local_renaming
     | local_aliases COMMA local_renaming
       ;

local_renaming :
       IDENTIFIER
     | IDENTIFIER EQUAL IDENTIFIER
```

```
       ;

loop_composition :
       LOOP opt_internal_statements opt_condition_handlers END
       ;

mapping_type :
       MAPPING opt_size_limit_restriction
          FROM type_specification
            TO type_specification
       ;

modified_primary_value :
       primary_value
     | modified_primary_value value_modifiers
     | modified_primary_value actual_condition_parameters
       ;

move_statement :
       MOVE removable_component component_destination
       ;

name_declaration :
       NAME local_aliases FROM IDENTIFIER
       ;

name_expression :
       IDENTIFIER
     | IDENTIFIER selector_list
       ;

new_dynamic_variable_component :
       INTO       name_expression
     | INTO   SET name_expression
     | BEFORE     name_expression
     | BEFORE SEQ name_expression
     | BEHIND     name_expression
     | BEHIND SEQ name_expression
       ;

new_statement :
       NEW expression new_dynamic_variable_component
       ;

non_validated_specification_expression :
       expression
     | proof_directive expression
     | OPEN_PAREN proof_directive expression CLOSE_PAREN
       ;

number :
       DIGIT_LIST
     | base DIGIT_LIST
       ;

opt_access_specification :
       empty
     | access_specification
       ;

opt_actual_condition_parameters :
       empty
     | actual_condition_parameters
       ;

opt_condition_handlers :
       empty
     | WHEN opt_handler_list
       ;
```

```
opt_default_initial_value_expression :
       empty
     | COLON_EQUAL expression
       ;

opt_each_clause :
       empty
     | EACH IDENTIFIER COLON type_specification COMMA
       ;

opt_handler_list :
       empty
     | handler_list
       ;

opt_entry_specification :
       empty
     | ENTRY non_validated_specification_expression SEMI_COLON
       ;

opt_exit_specification :
       empty
     | EXIT non_validated_specification_expression SEMI_COLON
     | EXIT conditional_exit_specification SEMI_COLON
       ;

opt_external_conditions :
       empty
     | UNLESS OPEN_PAREN COND identifier_list CLOSE_PAREN
       ;

opt_external_data_objects :
       empty
     | external_data_objects
       ;

opt_group_name :
       empty
     | COND
       ;

opt_internal_environment :
       empty
     | internal_environment
       ;

opt_internal_initial_value :
       empty
     | COLON_EQUAL expression
       ;

opt_internal_statements :
       empty
     | statement_list opt_semi_colon
     | PENDING opt_semi_colon
       ;

opt_keep_specification :
       empty
     | KEEP non_validated_specification_expression SEMI_COLON
       ;

opt_semi_colon :
       empty | SEMI_COLON
       ;

opt_size_limit_restriction :
       empty
     | OPEN_PAREN expression CLOSE_PAREN
```

```
       ;

pre_computable_label_expression :
       number
     | MINUS number
     | CHARACTER_VALUE
     | IDENTIFIER
       ;

primary_value :
       literal_value
     | set_or_sequence_value
     | ENTRY_VALUE
     | IDENTIFIER
     | if_expression
     | OPEN_PAREN expression CLOSE_PAREN
       ;

procedural_statement :
       assignment_statement
     | leave_statement
     | move_statement
     | new_statement
     | procedure_statement
     | remove_statement
     | signal_statement
       ;

procedure_body :
       PENDING
     | BEGIN
         external_operational_specification
         opt_internal_environment
         opt_keep_specification
         opt_internal_statements
       END
       ;

procedure_composition_rule :
       if_composition
     | case_composition
     | loop_composition
     | begin_composition
       ;

procedure_declaration :
        PROCEDURE IDENTIFIER
        external_data_objects opt_external_conditions EQUAL
        procedure_body
       ;

procedure_statement :
       IDENTIFIER arg_list opt_actual_condition_parameters
       ;

program_description : scope_declaration_list opt_semi_colon
       ;

proof_directive :
       PROVE
     | ASSUME
       ;

range_limits :
       expression DOT_DOT expression
       ;

range :
       OPEN_PAREN range_limits CLOSE_PAREN
```

```
        ;

record_type : RECORD OPEN_PAREN fields CLOSE_PAREN
        ;

removable_component :
        ELEMENT expression FROM SET name_expression
      | name_expression
        ;

remove_statement :
        REMOVE removable_component
        ;

scalar_type : OPEN_PAREN identifier_list CLOSE_PAREN
        ;

scope_declaration :
        SCOPE IDENTIFIER EQUAL
        BEGIN
          unit_or_name_declaration_list opt_semi_colon
        END
        ;

scope_declaration_list :
        scope_declaration
      | scope_declaration_list SEMI_COLON scope_declaration
        ;

selector_list :
        component_selectors
      | selector_list component_selectors
        ;

sequence_type :
        SEQUENCE opt_size_limit_restriction OF type_specification
        ;

set_type :
        SET opt_size_limit_restriction OF type_specification
        ;

set_or_seq_mark :
        SET COLON
      | SEQ COLON
        ;

set_or_sequence_value :
        OPEN_PAREN set_or_seq_mark element_list CLOSE_PAREN
      | range
        ;

signal_statement :
        SIGNAL IDENTIFIER
        ;

similar_fields :
        identifier_list COLON type_specification
        ;

similar_formal_data_parameters :
        opt_access_specification identifier_list COLON type_specification
        ;

specification_expression :
        non_validated_specification_expression
      | validated_specification_expression
      | OPEN_PAREN validated_specification_expression CLOSE_PAREN
        ;
```

```
statement_list :
       statement
     | statement_list SEMI_COLON statement
       ;

statement :
       procedural_statement
     | procedure_composition_rule
     | assert_specification
       ;

type_declaration :
       TYPE IDENTIFIER EQUAL type_definition
       ;

type_definition :
       PENDING
     | scalar_type
     | array_type
     | record_type
     | set_type
     | sequence_type
     | mapping_type
     | type_specification opt_default_initial_value_expression
       ;

type_specification :
       IDENTIFIER
     | IDENTIFIER range
       ;

unit_declaration :
       type_declaration
     | procedure_declaration
     | function_declaration
     | constant_declaration
     | lemma_declaration
       ;

unit_or_name_declaration :
       unit_declaration
     | name_declaration
       ;

unit_or_name_declaration_list :
       unit_or_name_declaration
     | unit_or_name_declaration_list SEMI_COLON unit_or_name_declaration
       ;

validated_specification_expression :
       non_validated_specification_expression OTHERWISE IDENTIFIER
       ;

value_alterations :
       WITH OPEN_PAREN component_alterations_list CLOSE_PAREN
       ;

value_list :
       expression
     | value_list COMMA expression
       ;

value_modifiers :
       component_selectors
     | range
     | value_alterations
       ;
%%
#include "lex.yy.c"
```

Appendix B
Operator Precedence

The precedence levels of the Gypsy operators are:

```
 12     ALL SOME
 11     ->  IMP  IFF
 10     OR
  9     &  AND
  8     NOT
  7     =  EQ  NE  <  LT  LE  >  GT  GE  IN  SUB
  6     @  APPEND  UNION  INTERSECT  DIFFERENCE
  5     ADJOIN  OMIT
 4.5    :> (right associative)
  4     +  -(binary)  <:
  3     *  /  DIV  MOD
  2     -(unary)
  1     **
```

Operators with lower numbered precedence levels are performed first. All operators are left associative except for **:>**. A new precedence level of 4.5 also has been introduced for **:>** because Yacc does not allow both left and right associative operators at the same level.

Appendix C
Tagged Grammar

```
<access_specification, a> ::= CONST

    access(a) = 'const


<access_specification, a> ::= VAR

    access(a) = 'var


<actual_condition_parameters, cp> ::=
        UNLESS OPEN_PAREN <opt_group_name, g> <identifier_list, is>
        CLOSE_PAREN

    actual_cargs(cp) = id_list(is)


<arg_list, as> ::= OPEN_PAREN <value_list, vs> CLOSE_PAREN

             actual_dargs(as) = actual_dargs(vs)

                 arg_list(as) = as

                arg_listp(as) = t

           gf_adp(as,c,v,n,x) = gf_adp(vs,c,v,n,x)

     gf_selectors(as,c,v,n,x) = gf_adp(vs,c,v,n,x)

          gpf_adp(as,c,s,n,x) = gpf_adp(vs, c, s, n - 1, x)

    gpf_selectors(as,c,s,n,x) = gpf_adp(vs, c, s, n - 1, x)


<array_type, a> ::=
        ARRAY OPEN_PAREN <type_specification, it> CLOSE_PAREN OF
        <type_specification, ct>

    type_desc(a,sn,ut,x) = array_desc
                             (type_desc
                                (subtree_i(a, 'type_specification, 1), sn, ut,
                                 x),
                              type_desc
                                (subtree_i(a, 'type_specification, 2), sn, ut,
                                 x))
```

```
<assert_specification, s> ::= ASSERT <specification_expression, e>

    gp(s,c,s~,n,x) = gp(e, c, s~, n - 1, x)


<assignment_statement, s> ::= <name_expression, n> COLON_EQUAL <expression, e>

    gp(s,c,s~,n~,x) = gp_assign
                        (gp_parg(n, c, s~, n~ - 1, x),
                         gpf(e, c, s~, n~ - 1, x), s~, c, n~, x)


<base, b> ::= BINARY

    ibase(b) = 2


<base, b> ::= DECIMAL

    ibase(b) = 10


<base, b> ::= HEX

    ibase(b) = 16


<base, b> ::= OCTAL

    ibase(b) = 8


<begin_composition, s> ::=
        BEGIN <opt_internal_statements, ss> <opt_condition_handlers, c> END

    gp(s,c~,s~,n,x) = gp_cond(c, c~, gp(ss, c~, s~, n - 1, x), n - 1, x)


<binary_operator, op> ::= ADJOIN

            apply_binary_op(op,v1,v2) = gadjoin(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gadjoin(sv1,sv2,s0)
```

```
<binary_operator, op> ::= AND

          apply_binary_op(op,v1,v2) = gand(v1,v2)
   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gand(sv1,sv2,s0)


<binary_operator, op> ::= APPEND

          apply_binary_op(op,v1,v2) = gappend(v1,v2)
   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gappend(sv1,sv2,s0)


<binary_operator, op> ::= COLON_GT

          apply_binary_op(op,v1,v2) = gcons(v1,v2)
   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gcons(sv1,sv2,s0)


<binary_operator, op> ::= DIFFERENCE

          apply_binary_op(op,v1,v2) = gdifference(v1,v2)
   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gdifference(sv1,sv2,s0)


<binary_operator, op> ::= DIV

          apply_binary_op(op,v1,v2) = gdiv(v1,v2)
   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gdiv(sv1,sv2,s0)


<binary_operator, op> ::= EQ

          apply_binary_op(op,v1,v2) = gequal(v1,v2)
                   eq_opp(op,etype) = equality_typep(etype)
   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gequal(sv1,sv2,s0)


<binary_operator, op> ::= EQUAL

          apply_binary_op(op,v1,v2) = gequal(v1,v2)
                   eq_opp(op,etype) = equality_typep(etype)
```

```
   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gequal(sv1,sv2,s0)



<binary_operator, op> ::= GE

            apply_binary_op(op,v1,v2) = gge(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gge(sv1,sv2,s0)



<binary_operator, op> ::= GT

            apply_binary_op(op,v1,v2) = ggt(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_ggt(sv1,sv2,s0)



<binary_operator, op> ::= IFF

            apply_binary_op(op,v1,v2) = giff(v1,v2)

                     eq_opp(op,etype) = boolean_typep(etype)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_giff(sv1,sv2,s0)



<binary_operator, op> ::= IMP

            apply_binary_op(op,v1,v2) = gimp(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gimp(sv1,sv2,s0)



<binary_operator, op> ::= IN

            apply_binary_op(op,v1,v2) = gin(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gin(sv1,sv2,s0)



<binary_operator, op> ::= INTERSECT

            apply_binary_op(op,v1,v2) = gintersect(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gintersect(sv1,sv2,s0)



<binary_operator, op> ::= LE
```

```
          apply_binary_op(op,v1,v2) = gle(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gle(sv1,sv2,s0)


<binary_operator, op> ::= LT

          apply_binary_op(op,v1,v2) = glt(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_glt(sv1,sv2,s0)


<binary_operator, op> ::= LT_COLON

          apply_binary_op(op,v1,v2) = grcons(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_grcons(sv1,sv2,s0)


<binary_operator, op> ::= MINUS

          apply_binary_op(op,v1,v2) = gsubtract(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gsubtract(sv1,sv2,s0)


<binary_operator, op> ::= MOD

          apply_binary_op(op,v1,v2) = gmod(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gmod(sv1,sv2,s0)


<binary_operator, op> ::= NE

          apply_binary_op(op,v1,v2) = gne(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gne(sv1,sv2,s0)


<binary_operator, op> ::= OMIT

          apply_binary_op(op,v1,v2) = gomit(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gomit(sv1,sv2,s0)


<binary_operator, op> ::= OR
```

```
            apply_binary_op(op,v1,v2) = gor(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gor(sv1,sv2,s0)



<binary_operator, op> ::= PLUS

            apply_binary_op(op,v1,v2) = gplus(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gplus(sv1,sv2,s0)



<binary_operator, op> ::= SLASH

            apply_binary_op(op,v1,v2) = gquotient(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gquotient(sv1,sv2,s0)



<binary_operator, op> ::= STAR

            apply_binary_op(op,v1,v2) = gtimes(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gtimes(sv1,sv2,s0)



<binary_operator, op> ::= STAR_STAR

            apply_binary_op(op,v1,v2) = gpower(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gpower(sv1,sv2,s0)



<binary_operator, op> ::= SUB

            apply_binary_op(op,v1,v2) = gsub(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gsub(sv1,sv2,s0)



<binary_operator, op> ::= UNION

            apply_binary_op(op,v1,v2) = gunion(v1,v2)

   gpf_apply_binary_op(op,sv1,sv2,s0) = gpf_gunion(sv1,sv2,s0)



<bound_expression, b> ::=
        <identifier_list, q> COLON <type_specification, s> COMMA
```

```
        <expression, e>


    bound_boolean_expression(b) = e

                    bound_id(b) = bound_id(q)

               bound_id_type(b) = s

            bound_values(b,c,x) = marked_typed_value_set(type_desc(s,c,nil,x))

        cdr_quantified_names(b) = cdr_quantified_names(q)




<case_composition, s> ::=
        CASE <expression, e> <case_composition_body, b>
        <opt_condition_handlers, c> END


     case_labels(s) = case_labels(b)

    gp(s,c~,s~,n,x) = let k = gpf(e, c~, s~, n - 1, x)
                      let r = gp_case_label_check
                                (k,
                                 gpf_list(case_labels(s), c~, s~, n - 1, x),
                                 s~)
                      gp_cond
                        (c, c~,
                              if normal_state(r)
                              then gp_case_body(k, b, c~, s~, n - 1, x)
                         else r, n - 1, x)




<case_composition_body, b> ::= empty


               case_labels(b) = nil

    gp_case_body(k,b,c,s,n,x) = s




<case_composition_body, b> ::= ELSE COLON <opt_internal_statements, ss>


               case_labels(b) = nil

    gp_case_body(k,b,c,s,n,x) = gp(ss, c, s, n - 1, x)




<case_composition_body, b> ::=
        IS <case_labels, cs> COLON <opt_internal_statements, ss>
        <case_composition_body, b2>


               case_labels(b) = case_labels(cs) @ case_labels(b2)

    gp_case_body(k,b,c,s,n,x) = let s1 = gpf_gin
                                           (k,
                                            gpf_gset
                                              (gpf_list
                                                 (case_labels(b), c, s, n - 1,
```

```
                                   x),
                                base_type(type(result~(k))), s),
                             s)
                      if normal_state(s1)
                      then      if gtruep(result~(s1))
                                then gp(ss, c, s, n - 1, x)
                           else gp_case_body
                                  (k, b2, c, s, n - 1, x)
                 else s1
```

```
<case_exit, ce> ::=
        IS <case_exit_labels, l> COLON
        <non_validated_specification_expression, e>

    case_exit_list(ce,c) = case_exit_list2(l,e,c)

         exit_labels(ce) = exit_labels(l)
```

```
<case_exit_body, b> ::= <case_exit, c>

    case_exit_list(b,c~) = case_exit_list(c,c~)

          exit_labels(b) = exit_labels(c)
```

```
<case_exit_body, b> ::= <case_exit_body, b2> SEMI_COLON <case_exit, c>

    case_exit_list(b,c~) = case_exit_list(b2,c~) @ case_exit_list(c,c~)

          exit_labels(b) = exit_labels(b2) @ exit_labels(c)
```

```
<case_exit_labels, ls> ::= <exit_label, l>

    case_exit_list2(ls,e,c) = case_exit_list2(l,e,c)

            exit_labels(ls) = exit_labels(l)
```

```
<case_exit_labels, ls> ::= <case_exit_labels, ls2> COMMA <exit_label, l>

    case_exit_list2(ls,e,c) =   case_exit_list2(ls2,e,c)
                              @ case_exit_list2(l,e,c)

            exit_labels(ls) = exit_labels(ls2) @ exit_labels(l)
```

```
<case_labels, cs> ::= <pre_computable_label_expression, e>
```

```
    case_labels(cs) = rcons(nil,e)
```

```
<case_labels, cs> ::=
        <case_labels, cs2> COMMA <pre_computable_label_expression, e>

    case_labels(cs) = rcons(case_labels(cs2), e)
```

```
<component_alterations, as> ::= <opt_each_clause, e> <component_assignment, a>

      gf_modifiers(bv,as,c,v,n,x) =      if each_clausep(e)
                                         then let vs = bound_values(e,c,x)
                                                   if errorp(vs)
                                                   then marked
                                                          (vs,
                                                           default_value
                                                             (base_type
                                                                (type(bv))))
                                              else gf_each
                                                     (bound_id(e), vs, bv, a,
                                                      c, v, n, x)
                                    else gf_modifiers(bv,a,c,v,n,x)

    gpf_modifiers(sbv,as,c,s,n,x) =      if each_clausep(e)
                                         then let svs = gpf_bound_values
                                                          (e,c,s,x)
                                                   if normal_state(svs)
                                                   then gpf_each
                                                          (bound_id(e),
                                                           result~(svs), sbv,
                                                           a, c, s, n - 1, x)
                                              else svs
                                    else gpf_modifiers(sbv, a, c, s, n - 1, x)
```

```
<component_alterations, as> ::= <opt_each_clause, e> <component_creation, c>

      gf_modifiers(bv,as,c~,v,n,x) =      if each_clausep(e)
                                          then let vs = bound_values(e,c~,x)
                                                    if errorp(vs)
                                                    then marked
                                                           (vs,
                                                            default_value
                                                              (base_type
                                                                 (type(bv))))
                                               else gf_each
                                                      (bound_id(e), vs, bv, c,
                                                       c~, v, n, x)
                                     else gf_modifiers(bv,c,c~,v,n,x)

    gpf_modifiers(sbv,as,c~,s,n,x) =      if each_clausep(e)
                                          then let svs = gpf_bound_values
                                                           (e,c~,s,x)
                                                    if normal_state(svs)
                                                    then gpf_each
                                                           (bound_id(e),
                                                            result~(svs), sbv,
                                                            c, c~, s, n - 1,
                                                            x)
```

```
                                        else svs
                              else gpf_modifiers
                                     (sbv, c, c~, s, n - 1, x)
```

```
<component_alterations, as> ::= <opt_each_clause, e> <component_deletion, d>

      gf_modifiers(bv,as,c,v,n,x) =      if each_clausep(e)
                                         then let vs = bound_values(e,c,x)
                                                   if errorp(vs)
                                                   then marked
                                                          (vs,
                                                           default_value
                                                             (base_type
                                                                (type(bv))))
                                              else gf_each
                                                     (bound_id(e), vs, bv, d,
                                                      c, v, n, x)
                                    else gf_modifiers(bv,d,c,v,n,x)

    gpf_modifiers(sbv,as,c,s,n,x) =      if each_clausep(e)
                                         then let svs = gpf_bound_values
                                                          (e,c,s,x)
                                                   if normal_state(svs)
                                                   then gpf_each
                                                          (bound_id(e),
                                                           result~(svs), sbv,
                                                           d, c, s, n - 1, x)
                                              else svs
                                    else gpf_modifiers(sbv, d, c, s, n - 1, x)
```

```
<component_alterations_list, al> ::= <component_alterations, a>

      gf_modifiers(bv,al,c,v,n,x) = gf_modifiers(bv,a,c,v,n,x)

    gpf_modifiers(sbv,al,c,s,n,x) = gpf_modifiers(sbv, a, c, s, n - 1, x)
```

```
<component_alterations_list, al> ::=
        <component_alterations_list, al2> SEMI_COLON
        <component_alterations, a>

      gf_modifiers(bv,al,c,v,n,x) = gf_modifiers
                                      (gf_modifiers(bv,al2,c,v,n,x), a, c, v,
                                       n, x)

    gpf_modifiers(sbv,al,c,s,n,x) = gpf_modifiers
                                      (gpf_modifiers(sbv, al2, c, s, n - 1, x),
                                       a, c, s, n - 1, x)
```

```
<component_assignment, a> ::= <selector_list, s> COLON_EQUAL <expression, e>

       gf_modifiers(bv,a,c,v,n,x) = put_op
                                      (bv, gf_selectors(s,c,v,n,x),
                                       gf(e,c,v,n,x))
```

```
    gpf_modifiers(sbv,a,c,s~,n,x) = gpf_put_op
                                     (sbv, gpf_selectors(s, c, s~, n - 1, x),
                                      gpf(e, c, s~, n - 1, x), s~)




<component_creation, c> ::=
        BEFORE <selector_list, s> COLON_EQUAL <expression, e>


       gf_modifiers(bv,c,c~,v,n,x) = let s~ = gf_selectors(s,c~,v,n,x)
                                        u = gf(e,c~,v,n,x)
                                    put_op
                                      (bv, rcdr(s~),
                                       gseq_insert_before
                                         (select_op(bv, rcdr(s~)), rcar(s~),
                                          u))

    gpf_modifiers(sbv,c,c~,s~,n,x) = let ss = gpf_selectors
                                                (s, c~, s~, n - 1, x)
                                        su = gpf(e, c~, s~, n - 1, x)
                                    gpf_put_op
                                      (sbv, rcdr(ss),
                                       gpf_gseq_insert_before
                                         (gpf_select_op(sbv, rcdr(ss), s~),
                                          rcar(ss), su, s~), s~)




<component_creation, c> ::=
        BEHIND <selector_list, s> COLON_EQUAL <expression, e>


       gf_modifiers(bv,c,c~,v,n,x) = let s~ = gf_selectors(s,c~,v,n,x)
                                        u = gf(e,c~,v,n,x)
                                    put_op
                                      (bv, rcdr(s~),
                                       gseq_insert_behind
                                         (select_op(bv, rcdr(s~)), rcar(s~),
                                          u))

    gpf_modifiers(sbv,c,c~,s~,n,x) = let ss = gpf_selectors
                                                (s, c~, s~, n - 1, x)
                                        su = gpf(e, c~, s~, n - 1, x)
                                    gpf_put_op
                                      (sbv, rcdr(ss),
                                       gpf_gseq_insert_behind
                                         (gpf_select_op(sbv, rcdr(ss), s~),
                                          rcar(ss), su, s~), s~)




<component_creation, c> ::=
        INTO <selector_list, s> COLON_EQUAL <expression, e>


       gf_modifiers(bv,c,c~,v,n,x) = let s~ = gf_selectors(s,c~,v,n,x)
                                        u = gf(e,c~,v,n,x)
                                    put_op
                                      (bv, rcdr(s~),
                                       gmap_insert
                                         (select_op(bv, rcdr(s~)), rcar(s~),
                                          u))

    gpf_modifiers(sbv,c,c~,s~,n,x) = let ss = gpf_selectors
                                                (s, c~, s~, n - 1, x)
```

```
                                        su = gpf(e, c~, s~, n - 1, x)
                                    gpf_put_op
                                      (sbv, rcdr(ss),
                                       gpf_gmap_insert
                                         (gpf_select_op(sbv, rcdr(ss), s~),
                                          rcar(ss), su, s~), s~)
```

```
<component_deletion, d> ::= MAPOMIT <selector_list, s>


       gf_modifiers(bv,d,c,v,n,x) = let s~ = gf_selectors(s,c,v,n,x)
                                    put_op
                                      (bv, rcdr(s~),
                                       gmapomit
                                         (select_op(bv, rcdr(s~)), rcar(s~)))

    gpf_modifiers(sbv,d,c,s~,n,x) = let ss = gpf_selectors(s, c, s~, n - 1, x)
                                    gpf_put_op
                                      (sbv, rcdr(ss),
                                       gpf_gmapomit
                                         (gpf_select_op(sbv, rcdr(ss), s~),
                                          rcar(ss), s~), s~)
```

```
<component_deletion, d> ::= SEQOMIT <selector_list, s>


       gf_modifiers(bv,d,c,v,n,x) = let s~ = gf_selectors(s,c,v,n,x)
                                    put_op
                                      (bv, rcdr(s~),
                                       gseqomit
                                         (select_op(bv, rcdr(s~)), rcar(s~)))

    gpf_modifiers(sbv,d,c,s~,n,x) = let ss = gpf_selectors(s, c, s~, n - 1, x)
                                    gpf_put_op
                                      (sbv, rcdr(ss),
                                       gpf_gseqomit
                                         (gpf_select_op(sbv, rcdr(ss), s~),
                                          rcar(ss), s~), s~)
```

```
<component_destination, d> ::= <new_dynamic_variable_component, dc>


    assign_dynamic_name(d,nne) = assign_dynamic_name(dc,nne)

        gmove_assign(d,v,ne,s) = gnew0(dc,v,ne,s)

               new_name_arg(d) = new_name_arg(dc)
```

```
<component_destination, d> ::= TO <name_expression, ne>


    assign_dynamic_name(d,nne) = name_exp
                                   (ne_name(nne), rcdr(ne_selectors(nne)))

       gmove_assign(d,v,ne~,s) =      if sequence_descp
                                           (type
                                              (stored_value
```

```
                                                  (name_exp
                                                     (ne_name(ne~),
                                                      rcdr(ne_selectors(ne~))),
                                                   s))) then gassign0(ne~,v,s)
                                 else set_condition(s, 'routineerror)

               new_name_arg(d) = ne
```

```
<component_selectors, s> ::= <arg_list, as>

                      arg_list(s) = as

                     arg_listp(s) = t

       gf_modifiers(bv,s,c,v,n,x) = select_op(bv, gf_adp(as,c,v,n,x))

          gf_selectors(s,c,v,n,x) = gf_adp(as,c,v,n,x)

    gpf_modifiers(sbv,s,c,s~,n,x) = gpf_select_op
                                      (sbv, gpf_adp(as, c, s~, n - 1, x), s~)

        gpf_selectors(s,c,s~,n,x) = gpf_adp(as, c, s~, n - 1, x)
```

```
<component_selectors, s> ::= DOT <IDENTIFIER, fn>

       gf_modifiers(bv,s,c,v,n,x) = record_get
                                      (bv, marked('field_name, gname(fn)),
                                       type(bv))

          gf_selectors(s,c,v,n,x) = list(marked('field_name, gname(fn)))

    gpf_modifiers(sbv,s,c,s~,n,x) = gpf_record_get
                                      (sbv,
                                       allocate
                                         ('result~,
                                          marked('field_name, gname(fn)), s~),
                                       s~)

        gpf_selectors(s,c,s~,n,x) = list
                                      (allocate
                                         ('result~,
                                          marked('field_name, gname(fn)), s~))
```

```
<conditional_exit_specification, c> ::=
        CASE OPEN_PAREN <case_exit_body, e> CLOSE_PAREN

    case_exit_list(c,c~) = case_exit_list(e,c~)

          exit_labels(c) = exit_labels(e)
```

```
<constant_body, b> ::= PENDING

    constant_value_exp(b) = nil
```

```
          gpf(b,c,s,n,x) = mark_state_indeterminate(s)



<constant_body, b> ::= <expression, p>

    constant_value_exp(b) = p

           gf(b,c,v,n,x) = gf(p,c,v,n,x)

          gpf(b,c,s,n,x) = gpf(p, c, s, n - 1, x)



<constant_declaration, d> ::=
        CONST <IDENTIFIER, cn> COLON <type_specification, rt> COLON_EQUAL
        <constant_body, b>

         constant_body(d) = b

    constant_value_exp(d) = constant_value_exp(b)

          formal_dargs(d) = nil

                  kind(d) = 'constant

                  prec(d) = mk_true_expression

           result_type(d) = rt

             unit_list(d) = cons(d,nil)

             unit_name(d) = unit_name(cn)



<element_list, e> ::= <range_limits, r>

     gf_element_list(e,c,v,n,x) = gf_element_list(r,c,v,n,x)

     gf_element_type(e,c,v,n,x) = gf_element_type(r,c,v,n,x)

    gpf_element_list(e,c,s,n,x) = gpf_element_list(r, c, s, n - 1, x)

    gpf_element_type(e,c,s,n,x) = gpf_element_type(r, c, s, n - 1, x)



<element_list, e> ::= <value_list, v>

    gf_element_list(e,c,v~,n,x) = gf_element_list(v,c,v~,n,x)

    gf_element_type(e,c,v~,n,x) = gf_element_type(v,c,v~,n,x)

    gpf_element_list(e,c,s,n,x) = gpf_element_list(v, c, s, n - 1, x)

    gpf_element_type(e,c,s,n,x) = gpf_element_type(v, c, s, n - 1, x)
```

```
<exit_label, l> ::= NORMAL


    case_exit_list2(l,e,c) = if c = 'normal then list(e) else nil

            exit_labels(l) = list('normal)



<exit_label, l> ::= <IDENTIFIER, n>


    case_exit_list2(l,e,c) = case_exit_list2(n,e,c)

            exit_labels(l) = exit_labels(n)



<expression, e> ::= <modified_primary_value, m>


              arg_list(e) = arg_list(m)

         fn_call_formp(e) = fn_call_formp(m)

            gf(e,c,v,n,x) = gf(m,c,v,n,x)

           gpf(e,c,s,n,x) = gpf(m, c, s, n - 1, x)

    mk_name_expression(e) = mk_name_expression(m)

           object_name(e) = object_name(m)

          object_namep(e) = object_namep(m)



<expression, e> ::= ALL <bound_expression, b>


    bound_boolean_expression(e) = bound_boolean_expression(b)

                    bound_id(e) = bound_id(b)

               bound_id_type(e) = bound_id_type(b)

            bound_values(e,c,x) = bound_values(b,c,x)

        cdr_quantified_names(e) = cdr_quantified_names(b)

                  gf(e,c,v,n,x) = let vs = bound_values(e,c,x)
                                       if errorp(vs)
                                       then marked
                                              (vs,
                                               default_value(boolean_desc))
                                  else gf_all
                                         (bound_id(b), vs,
                                          cdr_quantified_exp(e), c, v, n, x)

                 gpf(e,c,s,n,x) = let svs = gpf_bound_values(e,c,s,x)
                                       if normal_state(svs)
                                       then gpf_all
                                              (bound_id(b), result~(svs),
                                               cdr_quantified_exp(e), c, s,
                                               n - 1, x)
                                  else svs
```

```
                 quantifier(e) = 'all



<expression, e> ::= SOME <bound_expression, b>


    bound_boolean_expression(e) = bound_boolean_expression(b)

                   bound_id(e) = bound_id(b)

              bound_id_type(e) = bound_id_type(b)

           bound_values(e,c,x) = bound_values(b,c,x)

       cdr_quantified_names(e) = cdr_quantified_names(b)

                 gf(e,c,v,n,x) = let vs = bound_values(e,c,x)
                                      if errorp(vs)
                                      then marked
                                             (vs,
                                              default_value(boolean_desc))
                                 else gf_some
                                        (bound_id(b), vs,
                                         cdr_quantified_exp(e), c, v, n, x)

                gpf(e,c,s,n,x) = let svs = gpf_bound_values(e,c,s,x)
                                      if normal_state(svs)
                                      then gpf_some
                                             (bound_id(b), result~(svs),
                                              cdr_quantified_exp(e), c, s,
                                              n - 1, x)
                                 else svs

                 quantifier(e) = 'some



<expression, e> ::= <expression, e1> <binary_operator, op> <expression, e2>


    function_defn(u,ftype) = let fc = f_of_formals
                                        (unit_name(u), formal_dargs(u))
                                 result = mk_expression('result)
                             let e = postc(u, 'normal)
                                  if eq_opp(op,ftype)
                                  then      if    [  subtree_i
                                                         (e, 'expression, 1)
                                                     = result]
                                                  or [  subtree_i
                                                         (e, 'expression, 1)
                                                     = fc]
                                            then subst_tree
                                                    (fc, result,
                                                     subtree_i
                                                       (e, 'expression, 2))
                                       else if    [  subtree_i
                                                         (e, 'expression, 2)
                                                     = result]
                                                  or [  subtree_i
                                                         (e, 'expression, 2)
                                                     = fc]
                                            then subst_tree
                                                    (fc, result,
                                                     subtree_i
                                                       (e, 'expression, 1))
                                       else nil
```

```
                             else nil

             gf(e,c,v,n,x) = apply_binary_op
                               (op,
                                gf(subtree_i(e, 'expression, 1), c, v, n, x),
                                gf(subtree_i(e, 'expression, 2), c, v, n, x))

            gpf(e,c,s,n,x) = gpf_apply_binary_op
                               (op,
                                gpf
                                  (subtree_i(e, 'expression, 1), c, s, n - 1,
                                   x),
                                gpf
                                  (subtree_i(e, 'expression, 2), c, s, n - 1,
                                   x), s)



<expression, e> ::= <unary_operator, op> <expression, e2>


     gf(e,c,v,n,x) = apply_unary_op(op, gf(e2,c,v,n,x))

    gpf(e,c,s,n,x) = gpf_apply_unary_op(op, gpf(e2, c, s, n - 1, x), s)



<external_data_objects, d> ::=
        OPEN_PAREN <external_data_objects_list, d2> CLOSE_PAREN


    formal_dargs(d) = formal_dargs(d2)



<external_data_objects_list, d> ::= <similar_formal_data_parameters, d2>


    formal_dargs(d) = formal_dargs(d2)



<external_data_objects_list, d> ::=
        <external_data_objects_list, d2> SEMI_COLON
        <similar_formal_data_parameters, d3>


    formal_dargs(d) = formal_dargs(d2) @ formal_dargs(d3)



<external_operational_specification, s> ::=
        <opt_entry_specification, e> <opt_exit_specification, x>


    case_exit_list(s,c) = case_exit_list(x,c)

           exit_spec(s) = x

                prec(s) = prec(e)
```

```
<fields, f> ::= <similar_fields, s>


    field_descs(f,sn,ut,x) = field_descs(s,sn,ut,x)

     record_field_names(f) = record_field_names(s)



<fields, f> ::= <fields, f2> SEMI_COLON <similar_fields, s>


    field_descs(f,sn,ut,x) = let f1 = field_descs(f2,sn,ut,x)
                                 f2~ = field_descs(s,sn,ut,x)
                                  if error_descp(f1) then f1
                             else if error_descp(f2~) then f2~
                             else f1@f2~

     record_field_names(f) = record_field_names(f2) @ record_field_names(s)



<function_declaration, d> ::=
        FUNCTION <IDENTIFIER, fn> <opt_external_data_objects, a> COLON
        <type_specification, rt> <opt_external_conditions, c> EQUAL
        <procedure_body, b>


    case_exit_list(d,c~) = case_exit_list(b,c~)

            exit_spec(d) = exit_spec(b)

         formal_cargs(d) = formal_cargs(c)

         formal_dargs(d) = formal_dargs(a)

            keep_spec(d) = keep_spec(b)

                 kind(d) = 'function

                 prec(d) = prec(b)

       procedure_body(d) = b

          result_type(d) = rt

            unit_list(d) = cons(d,nil)

            unit_name(d) = unit_name(fn)



<handler, h> ::= IS <identifier_list, cs> COLON <opt_internal_statements, s>


         handler(h,c) = if member(c, id_list(cs)) then s else nil

    handler_labels(h) = id_list(cs)



<handler_list, hs> ::= <handler, h>


         handler(hs,c) = handler(h,c)
```

```
    handler_labels(hs) = handler_labels(h)



<handler_list, hs> ::= <handler_list, hs2> <handler, h>


         handler(hs,c) = let r = handler(hs2,c)
                         if r=nil then handler(h,c) else r

    handler_labels(hs) = handler_labels(hs2) @ handler_labels(h)



<IDENTIFIER, i> ::= <letter>{[_]<letter or digit>}


               bound_id(i) = gname(i)

    case_exit_list2(i,e,c) = if gname(i) = c then list(e) else nil

            dparam_name(i) = gname(i)

            exit_labels(i) = list(gname(i))

           foreign_name(i) = gname(i)

     foreign_scope_name(i) = gname(i)

            gf(i,c,v,n,x) = gapply(gname(i), nil, c, v, n, x)

                  gname(i) = let n = uc_list(lexeme(i))
                             if pack(n) = nil then 'nil~ else pack(n)

           gpf(i,c,s,n,x) = gpf_apply(gname(i), nil, nil, c, s, n - 1, x)

                id_list(i) = gname(i)

             local_name(i) = gname(i)

     mk_name_expression(i) = mk_tree('name_expression, i)

            object_name(i) = gname(i)

     record_field_names(i) = gname(i)

             scope_name(i) = gname(i)

      type_desc(i,sn,ut,x) = let tn = gname(i)
                                  if tn = 'boolean then boolean_desc
                             else if tn = 'character then character_desc
                             else if tn = 'integer then integer_desc
                             else if tn = 'rational then rational_desc
                             else let r = ref(tn,sn,x)
                                  let h = ref_scope(r)
                                      u = ref_unit(r)
                                       if member(r,ut)
                                       then type_defn_cycle_error(tn,sn)
                                  else if kind(u) = 'type
                                       then type_desc(u, h, cons(r,ut), x)
                                  else if errorp(u) then u
                                  else not_type_error(tn,sn)

              unit_name(i) = gname(i)
```

```
<identifier_list, is> ::= <IDENTIFIER, i>


                  bound_id(is) = bound_id(i)

               dparam_name(is) = dparam_name(i)

           full_dargs(a,is,ft) = rcons
                                   (nil,
                                    mk_single_formal_data_parameter(a,i,ft))

                   id_list(is) = rcons(nil, id_list(i))

        record_field_names(is) = rcons(nil, record_field_names(i))

    scalar_const_units(n,is,q) = rcons
                                   (nil, mk_scalar_const_unit(n, gname(i), q))

         scalar_value_list(is) = rcons(nil, gname(i))



<identifier_list, is> ::= <identifier_list, is2> COMMA <IDENTIFIER, i>


                  bound_id(is) = bound_id(is2)

      cdr_quantified_names(is) = mk_identifier_list
                                   (cdr_quantified_names(is2), i)

           full_dargs(a,is,ft) = rcons
                                   (full_dargs(a,is2,ft),
                                    mk_single_formal_data_parameter(a,i,ft))

                   id_list(is) = rcons(id_list(is2), id_list(i))

        record_field_names(is) = rcons
                                   (record_field_names(is2),
                                    record_field_names(i))

    scalar_const_units(n,is,q) = rcons
                                   (scalar_const_units(n, is2, q - 1),
                                    mk_scalar_const_unit(n, gname(i), q))

         scalar_value_list(is) = rcons(scalar_value_list(is2), gname(i))



<if_composition, s> ::=
        IF <expression, b> THEN <opt_internal_statements, ss>
        <if_composition_else_part, ep> <opt_condition_handlers, cs> END


               gp(s,c,s~,n,x) = let bv = gpf_type_check
                                           (boolean_desc,
                                            gpf(b, c, s~, n - 1, x))
                                    ep~ = if_statement_else_part(s)
                                gp_cond
                                  (cs, c,
                                        if normal_state(bv)
                                        then      if gtruep(result~(bv))
                                                  then gp(ss, c, s~, n - 1, x)
                                             else if ep~=nil then s~
                                             else gp(ep~, c, s~, n - 1, x)
                                   else bv, n - 1, x)

    if_statement_else_part(s) = if_statement_else_part(ep)
```

```
<if_composition_else_part, ep> ::= empty


    if_statement_else_part(ep) = nil



<if_composition_else_part, ep> ::=
        ELIF <expression, b> THEN <opt_internal_statements, ss>
        <if_composition_else_part, ep2>


    if_statement_else_part(ep) = mk_elif_into_if_statement(ep)



<if_composition_else_part, ep> ::= ELSE <opt_internal_statements, ss>


    if_statement_else_part(ep) = ss



<if_expression, i> ::=
        IF <expression, b> THEN <expression, p> <if_expression_else_part, e>


     gf(i,c,v,n,x) = let bv = gf(subtree_i(i, 'expression, 1), c, v, n, x)
                          if indeterminate(bv)
                          then marked
                                 (mark(bv),
                                  default_value
                                    (type
                                       (gf
                                          (subtree_i(i, 'expression, 2), c, v,
                                           n, x))))
                     else if truep(in_type(boolean_desc, bv))
                          then      if gtruep(bv)
                                    then gf
                                           (subtree_i(i, 'expression, 2), c,
                                            v, n, x)
                               else gf(if_else_exp(e), c, v, n, x)
                     else marked
                            (if_test_not_boolean_error(i,c),
                             default_value
                               (type
                                  (gf
                                     (subtree_i(i, 'expression, 2), c, v, n,
                                      x))))

    gpf(i,c,s,n,x) = let bv = gpf_type_check
                                (boolean_desc,
                                 gpf
                                   (subtree_i(i, 'expression, 1), c, s, n - 1,
                                    x))
                          if not normal_state(bv) then bv
                     else if gtruep(result~(bv))
                          then gpf
                                 (subtree_i(i, 'expression, 2), c, s, n - 1,
                                  x)
                     else gpf(if_else_exp(e), c, s, n - 1, x)

    if_else_exp(i) = if_else_exp(e)
```

```
<if_expression_else_part, e> ::=
        ELIF <expression, b> THEN <expression, p>
        <if_expression_else_part, e2>


    if_else_exp(e) = mk_tree
                       ('if_expression,
                        cons(mk_reserved_word('if), cdr(subtrees(e))))



<if_expression_else_part, e> ::= ELSE <expression, p> FI


    if_else_exp(e) = p



<internal_data_or_condition_objects, iv> ::=
        COND <identifier_list, is> SEMI_COLON


    gp_locals(iv,c,s,n,x) = gp_local_conds(id_list(is), s)



<internal_data_or_condition_objects, iv> ::=
        <access_specification, a> <identifier_list, is> COLON
        <type_specification, ts> <opt_internal_initial_value, v> SEMI_COLON


                        access(iv) = access(a)

             gp_locals(iv,c,s,n,x) = let ie = internal_initial_value_exp(iv)
                                     let iv~ =      if ie=nil then ie
                                               else gpf(ie, c, s, n - 1, x)
                                     gp_bind_locals
                                       (access(iv), id_list(is),
                                        type_desc(ts,c,nil,x), iv~, s)

    internal_initial_value_exp(iv) = internal_initial_value_exp(v)



<internal_environment, iv> ::= <internal_data_or_condition_objects, iv2>


    gp_locals(iv,c,s,n,x) = gp_locals(iv2, c, s, n - 1, x)



<internal_environment, iv> ::=
        <internal_environment, iv2> <internal_data_or_condition_objects, iv3>


    gp_locals(iv,c,s,n,x) = gp_locals
                              (iv3, c, gp_locals(iv2, c, s, n - 1, x), n - 1,
                               x)



<leave_statement, s> ::= LEAVE
```

```
    gp(s,c,s~,n,x) = set_condition(s~,'leave)



<lemma_declaration, d> ::=
        LEMMA <IDENTIFIER, ln> <opt_external_data_objects, a> EQUAL
        <non_validated_specification_expression, b>


         kind(d) = 'lemma

    unit_list(d) = cons(d,nil)

    unit_name(d) = unit_name(ln)



<literal_value, l> ::= <CHARACTER_VALUE, ch>


     gf(l,c,v,n,x) = gf(ch,c,v,n,x)

    gpf(l,c,s,n,x) = gpf(ch, c, s, n - 1, x)



<literal_value, l> ::= <number, n>


     gf(l,c,v,n~,x) = gf(n,c,v,n~,x)

    gpf(l,c,s,n~,x) = gpf(n, c, s, n~ - 1, x)



<literal_value, l> ::= <STRING_VALUE, s>


      gf(l,c,v,n,x) = gf(s,c,v,n,x)

    gpf(l,c,s~,n,x) = gpf(s, c, s~, n - 1, x)



<local_aliases, a> ::= <local_renaming, r>


          foreign_name(a) = foreign_name(r)

            local_name(a) = local_name(r)

    named_unit_list(a,fs) = rcons(nil, named_unit(r,fs))



<local_aliases, a> ::= <local_aliases, a2> COMMA <local_renaming, r>


          foreign_name(a) = foreign_name(r)

            local_name(a) = local_name(r)

    named_unit_list(a,fs) = rcons(named_unit_list(a2,fs), named_unit(r,fs))
```

```
<local_renaming, r> ::= <IDENTIFIER, fn>


    foreign_name(r) = foreign_name(fn)

      local_name(r) = local_name(fn)



<local_renaming, r> ::= <IDENTIFIER, ln> EQUAL <IDENTIFIER, fn>


    foreign_name(r) = foreign_name(subtree_i(r, 'identifier, 2))

      local_name(r) = local_name(subtree_i(r, 'identifier, 1))



<loop_composition, s> ::=
        LOOP <opt_internal_statements, ss> <opt_condition_handlers, c> END


    gp(s,c~,s~,n,x) = let p1 = gp(ss, c~, s~, n - 1, x)
                      gp_cond
                        (c, c~,
                              if condition_non_normal(p1)
                              then reset_leave_to_normal(p1)
                         else gp(s, c~, p1, n - 1, x), n - 1, x)



<mapping_type, m> ::=
        MAPPING <opt_size_limit_restriction, r> FROM <type_specification, st>
        TO <type_specification, ct>


    type_desc(m,sn,ut,x) = mapping_desc
                             (size_limit(r,sn,x),
                              type_desc
                                (subtree_i(m, 'type_specification, 1), sn, ut,
                                 x),
                              type_desc
                                (subtree_i(m, 'type_specification, 2), sn, ut,
                                 x))



<modified_primary_value, m> ::= <primary_value, p>


         fn_call_formp(m) = object_namep(m)

            gf(m,c,v,n,x) = gf(p,c,v,n,x)

           gpf(m,c,s,n,x) = gpf(p, c, s, n - 1, x)

    mk_name_expression(m) = mk_name_expression(p)

           object_name(m) = object_name(p)

          object_namep(m) = object_namep(p)
```

```
<modified_primary_value, m> ::=
        <modified_primary_value, m2> <actual_condition_parameters, cp>


     actual_cargs(m) = actual_cargs(cp)

         arg_list(m) = arg_list(m2)

    fn_call_formp(m) = fn_call_formp(m2)

       gf(m,c,v,n,x) = let r = gf(m2,c,v,n,x)
                       marked
                         (condition_params_error(m), default_value(type(r)))

      gpf(m,c,s,n,x) =      if fn_call_formp(m)
                            then gpf_apply
                                   (object_name(m2),
                                    gpf_adp(arg_list(m2), c, s, n - 1, x),
                                    actual_cargs(m), c, s, n - 1, x)
                       else set_condition(s, 'routineerror)

      object_name(m) = object_name(m2)




<modified_primary_value, m> ::=
        <modified_primary_value, m2> <value_modifiers, vm>


              arg_list(m) = arg_list(vm)

         fn_call_formp(m) = object_namep(m2) & arg_listp(vm)

            gf(m,c,v,n,x) =      if fn_call_formp(m)
                                 then gapply
                                        (object_name(m2),
                                         gf_adp(arg_list(vm), c, v, n, x), c,
                                         v, n, x)
                            else gf_modifiers(gf(m2,c,v,n,x), vm, c, v, n, x)

           gpf(m,c,s,n,x) =      if fn_call_formp(m)
                                 then gpf_apply
                                        (object_name(m2),
                                         gpf_adp(arg_list(vm), c, s, n - 1, x),
                                         nil, c, s, n - 1, x)
                            else gpf_modifiers
                                   (gpf(m2, c, s, n - 1, x), vm, c, s, n - 1,
                                    x)

    mk_name_expression(m) = extend_name_selectors
                              (mk_name_expression(m2),
                               component_selectors(vm))

           object_name(m) = object_name(m2)




<move_statement, s> ::=
        MOVE <removable_component, c> <component_destination, d>


       gp(s,c~,s~,n,x) = let e = remove_exp_arg(s)
                         gp_move
                           (if e=nil then nil else gpf(e, c~, s~, n - 1, x),
                            gp_parg(remove_name_arg(s), c~, s~, n - 1, x), d,
                            gp_parg(new_name_arg(s), c~, s~, n - 1, x), c~,
                            s~, n, x)
```

```
       new_name_arg(s) = new_name_arg(d)

     remove_exp_arg(s) = remove_exp_arg(c)

    remove_name_arg(s) = remove_name_arg(c)



<name_declaration, d> ::= NAME <local_aliases, a> FROM <IDENTIFIER, fs>


          foreign_name(d) = foreign_name(a)

    foreign_scope_name(d) = foreign_scope_name(fs)

                  kind(d) = 'name

            local_name(d) = local_name(a)

             unit_list(d) = named_unit_list(a,fs)



<name_expression, e> ::= <IDENTIFIER, i>


    extend_name_selectors(e,cs) = mk_tree
                                    ('name_expression,
                                     list(i, mk_tree('selector_list, cs)))

            gp_parg(e~,c,s,n,x) = let e = mk_name_expression(e~)
                                  let vn = gname(i)
                                  let r = gpf_apply_var(vn,s,nil)
                                       if normal_state(r)
                                       then allocate
                                              ('result~, name_exp(vn,nil), s)
                                  else r



<name_expression, e> ::= <IDENTIFIER, i> <selector_list, ss>


    extend_name_selectors(e,cs) = mk_tree
                                    ('name_expression,
                                     list
                                       (i,
                                        mk_tree('selector_list, list(ss,cs))))

            gp_parg(e~,c,s,n,x) = let e = mk_name_expression(e~)
                                  let vn = gname(i)
                                      ss~ = gpf_selectors(ss, c, s, n - 1, x)
                                  let r = gpf_apply_var(vn,s,ss~)
                                       if normal_state(r)
                                       then allocate
                                              ('result~,
                                               name_exp(vn, result~_list(ss~)),
                                               s)
                                  else r



<new_dynamic_variable_component, dc> ::= BEFORE SEQ <name_expression, ne>
```

```
    gnew0(dc,v,ne~,s) = let id = ne_name(ne~)
                            ss = ne_selectors(ne~)
                        gassign0(ne~, gcons(v, apply_var(id, map(s), ss)), s)

     new_name_arg(dc) = ne
```

```
<new_dynamic_variable_component, dc> ::= BEFORE <name_expression, ne>
```

```
    assign_dynamic_name(dc,nne) = name_exp
                                    (ne_name(nne), rcdr(ne_selectors(nne)))

              gnew0(dc,v,ne~,s) = let id = ne_name(ne~)
                                      ss = ne_selectors(ne~)
                                  gassign0
                                    (name_exp(id, rcdr(ss)),
                                     gseq_insert_before
                                       (apply_var(id, map(s), rcdr(ss)),
                                        rcar(ss), v), s)

               new_name_arg(dc) = ne
```

```
<new_dynamic_variable_component, dc> ::= BEHIND SEQ <name_expression, ne>
```

```
    gnew0(dc,v,ne~,s) = let id = ne_name(ne~)
                            ss = ne_selectors(ne~)
                        gassign0(ne~, grcons(apply_var(id, map(s), ss), v), s)

     new_name_arg(dc) = ne
```

```
<new_dynamic_variable_component, dc> ::= BEHIND <name_expression, ne>
```

```
    assign_dynamic_name(dc,nne) = name_exp
                                    (ne_name(nne), rcdr(ne_selectors(nne)))

              gnew0(dc,v,ne~,s) = let id = ne_name(ne~)
                                      ss = ne_selectors(ne~)
                                  gassign0
                                    (name_exp(id, rcdr(ss)),
                                     gseq_insert_behind
                                       (apply_var(id, map(s), rcdr(ss)),
                                        rcar(ss), v), s)

               new_name_arg(dc) = ne
```

```
<new_dynamic_variable_component, dc> ::= INTO SET <name_expression, ne>
```

```
    gnew0(dc,v,ne~,s) = let id = ne_name(ne~)
                            ss = ne_selectors(ne~)
                        gassign0
                          (ne~, gadjoin(apply_var(id, map(s), ss), v), s)

     new_name_arg(dc) = ne
```

```
<new_dynamic_variable_component, dc> ::= INTO <name_expression, ne>


    assign_dynamic_name(dc,nne) = name_exp
                                    (ne_name(nne), rcdr(ne_selectors(nne)))

              gnew0(dc,v,ne~,s) = let id = ne_name(ne~)
                                      ss = ne_selectors(ne~)
                                  gassign0
                                    (name_exp(id, rcdr(ss)),
                                     gmap_insert
                                       (apply_var(id, map(s), rcdr(ss)),
                                        rcar(ss), v), s)

               new_name_arg(dc) = ne




<new_statement, s> ::=
        NEW <expression, e> <new_dynamic_variable_component, dc>


     gp(s,c,s~,n,x) = gp_new
                        (dc, gpf(e, c, s~, n - 1, x),
                         gp_parg(new_name_arg(s), c, s~, n - 1, x), c, s~, n,
                         x)

    new_name_arg(s) = new_name_arg(dc)




<non_validated_specification_expression, se> ::= <expression, e>


    expression_from_spec(se) = e

               keep_spec(se) = e

                    prec(se) = e




<non_validated_specification_expression, se> ::=
        OPEN_PAREN <proof_directive, d> <expression, e> CLOSE_PAREN


    expression_from_spec(se) = e

               keep_spec(se) = e

                    prec(se) = e




<non_validated_specification_expression, se> ::=
        <proof_directive, d> <expression, e>


    expression_from_spec(se) = e

               keep_spec(se) = e

                    prec(se) = e
```

```
<number, n> ::= <DIGIT_LIST, s>

      gf(n,c,v,n~,x) = minteger(n)

    gpf(n,c,s~,n~,x) = gpf_minteger(n,s~)

         minteger(n) = mdigit_value(s,10)


<number, n> ::= <base, b> <DIGIT_LIST, s>

      gf(n,c,v,n~,x) = minteger(n)

    gpf(n,c,s~,n~,x) = gpf_minteger(n,s~)

         minteger(n) = mdigit_value(s, ibase(b))


<opt_access_specification, a> ::= <access_specification, a2>

    access(a) = access(a2)


<opt_access_specification, a> ::= empty

    access(a) = 'const


<opt_actual_condition_parameters, cp> ::= <actual_condition_parameters, cp2>

    actual_cargs(cp) = actual_cargs(cp2)


<opt_actual_condition_parameters, cp> ::= empty

    actual_cargs(cp) = nil


<opt_condition_handlers, c> ::= empty

        handler(c,c~) = nil

    handler_labels(c) = nil


<opt_condition_handlers, c> ::= WHEN <opt_handler_list, hs>
```

```
        handler(c,c~) = handler(hs,c~)

    handler_labels(c) = handler_labels(hs)



<opt_default_initial_value_expression, v> ::= empty


    default_initial_value(v,sn,x) = nil



<opt_default_initial_value_expression, v> ::= COLON_EQUAL <expression, e>


    default_initial_value(v,sn,x) = precomputable_f(e,sn,x)



<opt_each_clause, e> ::= empty


    each_clausep(e) = f



<opt_each_clause, e> ::=
        EACH <IDENTIFIER, i> COLON <type_specification, ts> COMMA


            bound_id(e) = bound_id(i)

       bound_id_type(e) = ts

    bound_values(e,c,x) = let td = type_desc(ts,c,nil,x)
                               if bounded_index_typep(td)
                               then marked_typed_value_set(td)
                          else each_id_type_error(e,c)

        each_clausep(e) = t



<opt_entry_specification, e> ::= empty


    prec(e) = mk_true_expression



<opt_entry_specification, e> ::=
        ENTRY <non_validated_specification_expression, se> SEMI_COLON


    prec(e) = prec(se)



<opt_exit_specification, e> ::= empty
```

```
    case_exit_list(e,c) = nil

         exit_labels(e) = nil

           exit_spec(e) = e



<opt_exit_specification, e> ::=
        EXIT <conditional_exit_specification, c> SEMI_COLON


    case_exit_list(e,c~) = case_exit_list(c,c~)

          exit_labels(e) = exit_labels(c)

            exit_spec(e) = e



<opt_exit_specification, e> ::=
        EXIT <non_validated_specification_expression, se> SEMI_COLON


    case_exit_list(e,c) = if c = 'normal then list(se) else nil

         exit_labels(e) = nil

           exit_spec(e) = e



<opt_external_conditions, c> ::= empty


    formal_cargs(c) = nil



<opt_external_conditions, c> ::=
        UNLESS OPEN_PAREN COND <identifier_list, is> CLOSE_PAREN


    formal_cargs(c) = id_list(is)



<opt_external_data_objects, d> ::= empty


    formal_dargs(d) = nil



<opt_external_data_objects, d> ::= <external_data_objects, d2>


    formal_dargs(d) = formal_dargs(d2)



<opt_handler_list, hs> ::= empty
```

```
        handler(hs,c) = nil

    handler_labels(hs) = nil


<opt_handler_list, hs> ::= <handler_list, hs2>

        handler(hs,c) = handler(hs2,c)

    handler_labels(hs) = handler_labels(hs2)


<opt_internal_environment, iv> ::= empty

    gp_locals(iv,c,s,n,x) = s


<opt_internal_environment, iv> ::= <internal_environment, iv2>

    gp_locals(iv,c,s,n,x) = gp_locals(iv2, c, s, n - 1, x)


<opt_internal_initial_value, v> ::= empty

    internal_initial_value_exp(v) = nil


<opt_internal_initial_value, v> ::= COLON_EQUAL <expression, e>

    internal_initial_value_exp(v) = e


<opt_internal_statements, ss> ::= empty

    gp(ss,c,s,n,x) = s


<opt_internal_statements, ss> ::= PENDING opt_semi_colon

    gp(ss,c,s,n,x) = mark_state_indeterminate(s)


<opt_internal_statements, ss> ::= <statement_list, ss2> opt_semi_colon

    gp(ss,c,s,n,x) = gp(ss2, c, s, n - 1, x)
```

```
<opt_keep_specification, k> ::= empty


    keep_spec(k) = mk_true_expression



<opt_keep_specification, k> ::=
        KEEP <non_validated_specification_expression, se> SEMI_COLON


    keep_spec(k) = keep_spec(se)



<opt_size_limit_restriction, r> ::= empty


    size_limit(r,sn,x) = nil



<opt_size_limit_restriction, r> ::= OPEN_PAREN <expression, e> CLOSE_PAREN


    size_limit(r,sn,x) = let sl = precomputable_f(e,sn,x)
                              if determinate(sl)
                              then let ok = in_type(integer_desc, sl)
                                        if errorp(ok) then ok
                                   else if truep(ok) & [0 le value(sl)]
                                        then value(sl)
                                   else size_limit_error(r,sn)
                         else if errorp(mark(sl)) then mark(sl)
                         else mk_error(mark(sl))



<pre_computable_label_expression, p> ::= <CHARACTER_VALUE, ch>


     gf(p,c,v,n,x) = gf(ch,c,v,n,x)

    gpf(p,c,s,n,x) = gpf(ch, c, s, n - 1, x)



<pre_computable_label_expression, p> ::= <IDENTIFIER, i>


     gf(p,c,v,n,x) = gf(i,c,v,n,x)

    gpf(p,c,s,n,x) = gpf(i, c, s, n - 1, x)



<pre_computable_label_expression, p> ::= <number, n>


     gf(p,c,v,n~,x) = gf(n,c,v,n~,x)

    gpf(p,c,s,n~,x) = gpf(n, c, s, n~ - 1, x)
```

```
<pre_computable_label_expression, p> ::= MINUS <number, n>

     gf(p,c,v,n~,x) = apply_unary_op
                        (mk_unary_operator('minus), gf(n,c,v,n~,x))

    gpf(p,c,s,n~,x) = gpf_apply_unary_op
                        (mk_unary_operator('minus), gpf(n, c, s, n~ - 1, x),
                         s)



<primary_value, p> ::= <ENTRY_VALUE, e>

     gf(p,c,v,n,x) = gf(e,c,v,n,x)

    gpf(p,c,s,n,x) = gpf(e, c, s, n - 1, x)



<primary_value, p> ::= <IDENTIFIER, i>

            gf(p,c,v,n,x) = gf(i,c,v,n,x)

           gpf(p,c,s,n,x) = gpf(i, c, s, n - 1, x)

    mk_name_expression(p) = mk_name_expression(i)

           object_name(p) = object_name(i)

          object_namep(p) = t



<primary_value, p> ::= <if_expression, i>

     gf(p,c,v,n,x) = gf(i,c,v,n,x)

    gpf(p,c,s,n,x) = gpf(i, c, s, n - 1, x)



<primary_value, p> ::= <literal_value, l>

     gf(p,c,v,n,x) = gf(l,c,v,n,x)

    gpf(p,c,s,n,x) = gpf(l, c, s, n - 1, x)



<primary_value, p> ::= <set_or_sequence_value, s>

      gf(p,c,v,n,x) = gf(s,c,v,n,x)

    gpf(p,c,s~,n,x) = gpf(s, c, s~, n - 1, x)
```

```
<primary_value, p> ::= OPEN_PAREN <expression, e> CLOSE_PAREN

     gf(p,c,v,n,x) = gf(e,c,v,n,x)

    gpf(p,c,s,n,x) = gpf(e, c, s, n - 1, x)



<procedural_statement, s> ::= <assignment_statement, s2>

    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<procedural_statement, s> ::= <leave_statement, s2>

    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<procedural_statement, s> ::= <move_statement, s2>

    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<procedural_statement, s> ::= <new_statement, s2>

    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<procedural_statement, s> ::= <procedure_statement, s2>

    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<procedural_statement, s> ::= <remove_statement, s2>

    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<procedural_statement, s> ::= <signal_statement, s2>

    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<procedure_body, b> ::= PENDING
```

```
             case_exit_list(b,c) = nil

                    exit_spec(b) = nil

    gp_procedure_body(b,c,s,n,x) = mark_state_indeterminate(s)

                         prec(b) = mk_true_expression
```

```
<procedure_body, b> ::=
        BEGIN <external_operational_specification, es>
        <opt_internal_environment, iv> <opt_keep_specification, k>
        <opt_internal_statements, st> END


             case_exit_list(b,c) = case_exit_list(es,c)

                    exit_spec(b) = exit_spec(es)

    gp_procedure_body(b,c,s,n,x) = let r = gp_deallocate_locals
                                             (gp
                                                (st, c,
                                                 gp_set_keep
                                                   (keep_spec(b),
                                                    gp_locals
                                                      (iv, c,
                                                       gp_set_entry
                                                         (prec(b), c, s, n, x),
                                                       n - 1, x), c, n, x),
                                                 n - 1, x))
                                        if indeterminate(r) then r
                                   else if    [cond~(r) = 'normal]
                                           or conditionp(cond~(r), r)
                                        then gp_set_exit
                                               (exit_spec(b), c, r, n, x)
                                   else if cond~(r) = 'leave
                                        then set_condition(r, 'routineerror)
                                   else mark_state_indeterminate(r)

                    keep_spec(b) = keep_spec(k)

                         prec(b) = prec(es)
```

```
<procedure_composition_rule, s> ::= <begin_composition, s2>


    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)
```

```
<procedure_composition_rule, s> ::= <case_composition, s2>


    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)
```

```
<procedure_composition_rule, s> ::= <if_composition, s2>


    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)
```

```
<procedure_composition_rule, s> ::= <loop_composition, s2>


    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)




<procedure_declaration, d> ::=
        PROCEDURE <IDENTIFIER, pn> <external_data_objects, a>
        <opt_external_conditions, c> EQUAL <procedure_body, b>


    case_exit_list(d,c~) = case_exit_list(b,c~)

            exit_spec(d) = exit_spec(b)

         formal_cargs(d) = formal_cargs(c)

         formal_dargs(d) = formal_dargs(a)

            keep_spec(d) = keep_spec(b)

                 kind(d) = 'procedure

                 prec(d) = prec(b)

       procedure_body(d) = b

            unit_list(d) = cons(d,nil)

            unit_name(d) = unit_name(pn)




<procedure_statement, s> ::=
        <IDENTIFIER, pn> <arg_list, dp> <opt_actual_condition_parameters, cp>


    actual_cargs(s) = actual_cargs(cp)

    actual_dargs(s) = actual_dargs(dp)

     gp(s,c,s~,n,x) = gp_procedure_call
                        (gname(pn),
                         gp_parg_list(actual_dargs(s), c, s~, n - 1, x),
                         actual_cargs(s), c, s~, n - 1, x)




<program_description, pd> ::= <scope_declaration_list, ss> opt_semi_colon


    scope_list(pd) = scope_list(ss)




<range, r> ::= OPEN_PAREN <range_limits, r2> CLOSE_PAREN


      gf_element_list(r,c,v,n,x) = gf_element_list(r2,c,v,n,x)

      gf_element_type(r,c,v,n,x) = gf_element_type(r2,c,v,n,x)

      gf_modifiers(bv,r,c,v,n,x) = gf_modifiers(bv,r2,c,v,n,x)
```

```
     gpf_element_list(r,c,s,n,x) = gpf_element_list(r2, c, s, n - 1, x)

     gpf_element_type(r,c,s,n,x) = gpf_element_type(r2, c, s, n - 1, x)

    gpf_modifiers(sbv,r,c,s,n,x) = gpf_modifiers(sbv, r2, c, s, n - 1, x)

               range_max(r,sn,x) = range_max(r2,sn,x)

               range_min(r,sn,x) = range_min(r2,sn,x)
```

```
<range_limits, r> ::= <expression, lo> DOT_DOT <expression, hi>
```

```
      gf_element_list(r,c,v,n,x) = grange_elements
                                     (gf
                                        (subtree_i(r, 'expression, 1), c, v,
                                         n, x),
                                      gf
                                        (subtree_i(r, 'expression, 2), c, v,
                                         n, x))

      gf_element_type(r,c,v,n,x) = base_type
                                     (type
                                        (gf
                                           (subtree_i(r, 'expression, 1), c,
                                            v, n, x)))

      gf_modifiers(bv,r,c,v,n,x) = subsequence_get
                                     (bv,
                                      gf
                                        (subtree_i(r, 'expression, 1), c, v,
                                         n, x),
                                      gf
                                        (subtree_i(r, 'expression, 2), c, v,
                                         n, x))

     gpf_element_list(r,c,s,n,x) = gpf_grange_elements
                                     (gpf
                                        (subtree_i(r, 'expression, 1), c, s,
                                         n - 1, x),
                                      gpf
                                        (subtree_i(r, 'expression, 2), c, s,
                                         n - 1, x), s)

     gpf_element_type(r,c,s,n,x) = base_type
                                     (type
                                        (result~
                                           (gpf
                                              (subtree_i(r, 'expression, 1),
                                               c, s, n - 1, x))))

    gpf_modifiers(sbv,r,c,s,n,x) = gpf_subsequence_get
                                     (sbv,
                                      gpf
                                        (subtree_i(r, 'expression, 1), c, s,
                                         n - 1, x),
                                      gpf
                                        (subtree_i(r, 'expression, 2), c, s,
                                         n - 1, x), s)

               range_max(r,sn,x) = precomputable_f
                                     (subtree_i(r, 'expression, 2), sn, x)

               range_min(r,sn,x) = precomputable_f
                                     (subtree_i(r, 'expression, 1), sn, x)
```

```
<record_type, r> ::= RECORD OPEN_PAREN <fields, f> CLOSE_PAREN


    record_field_names(r) = record_field_names(f)

     type_desc(r,sn,ut,x) = record_desc(field_descs(f,sn,ut,x))



<removable_component, c> ::= <name_expression, e>


     remove_exp_arg(c) = nil

    remove_name_arg(c) = e



<removable_component, c> ::=
        ELEMENT <expression, e> FROM SET <name_expression, ne>


     remove_exp_arg(c) = e

    remove_name_arg(c) = ne



<remove_statement, s> ::= REMOVE <removable_component, c>


       gp(s,c~,s~,n,x) = let e = remove_exp_arg(s)
                         gp_remove
                           (if e=nil then nil else gpf(e, c~, s~, n - 1, x),
                            gp_parg(remove_name_arg(s), c~, s~, n - 1, x), c~,
                            s~, n, x)

     remove_exp_arg(s) = remove_exp_arg(c)

    remove_name_arg(s) = remove_name_arg(c)



<scalar_type, s> ::= OPEN_PAREN <identifier_list, is> CLOSE_PAREN


      derived_units(n,s) = scalar_const_units
                             (n, is, length(scalar_value_list(s)) - 1)

    scalar_value_list(s) = scalar_value_list(is)



<scope_declaration, sd> ::=
        SCOPE <IDENTIFIER, sn> EQUAL BEGIN <unit_or_name_declaration_list, ul>
        opt_semi_colon END


    scope_name(sd) = scope_name(sn)

     unit_list(sd) = unit_list(ul)
```

```
<scope_declaration_list, ss> ::= <scope_declaration, sd>


    scope_list(ss) = rcons(nil,sd)



<scope_declaration_list, ss> ::=
        <scope_declaration_list, ss2> SEMI_COLON <scope_declaration, sd>


    scope_list(ss) = rcons(scope_list(ss2), sd)



<selector_list, s> ::= <component_selectors, s2>


      gf_selectors(s,c,v,n,x) = gf_selectors(s2,c,v,n,x)

    gpf_selectors(s,c,s~,n,x) = gpf_selectors(s2, c, s~, n - 1, x)



<selector_list, s> ::= <selector_list, s2> <component_selectors, s3>


      gf_selectors(s,c,v,n,x) =   gf_selectors(s2,c,v,n,x)
                                @ gf_selectors(s3,c,v,n,x)

    gpf_selectors(s,c,s~,n,x) =   gpf_selectors(s2, c, s~, n - 1, x)
                                @ gpf_selectors(s3, c, s~, n - 1, x)



<sequence_type, s> ::=
        SEQUENCE <opt_size_limit_restriction, r> OF <type_specification, ct>


    type_desc(s,sn,ut,x) = sequence_desc
                             (size_limit(r,sn,x), type_desc(ct,sn,ut,x))



<set_or_sequence_value, s> ::= <range, r>


      gf(s,c,v,n,x) = gset_or_seq
                        (nil, gf_element_list(r,c,v,n,x),
                         gf_element_type(r,c,v,n,x))

    gpf(s,c,s~,n,x) = gpf_gset_or_seq
                        (nil, gpf_element_list(r, c, s~, n - 1, x),
                         gpf_element_type(r, c, s~, n - 1, x), s~)



<set_or_sequence_value, s> ::=
        OPEN_PAREN <set_or_seq_mark, m> <element_list, e> CLOSE_PAREN


      gf(s,c,v,n,x) = gset_or_seq
                        (m, gf_element_list(e,c,v,n,x),
```

```
                         gf_element_type(e,c,v,n,x))

    gpf(s,c,s~,n,x) = gpf_gset_or_seq
                        (m, gpf_element_list(e, c, s~, n - 1, x),
                         gpf_element_type(e, c, s~, n - 1, x), s~)
```

```
<set_or_seq_mark, m> ::= SEQ COLON
```

```
    gpf_gset_or_seq(m,ses,td,s0) = gpf_gseq(ses,td,s0)

             gset_or_seq(m,a,td) = gseq(a,td)
```

```
<set_or_seq_mark, m> ::= SET COLON
```

```
    gpf_gset_or_seq(m,ses,td,s0) = gpf_gset(ses,td,s0)

             gset_or_seq(m,a,td) = gset(a,td)
```

```
<set_type, s> ::=
        SET <opt_size_limit_restriction, r> OF <type_specification, ct>
```

```
    type_desc(s,sn,ut,x) = set_desc(size_limit(r,sn,x), type_desc(ct,sn,ut,x))
```

```
<signal_statement, s> ::= SIGNAL <IDENTIFIER, c>
```

```
    gp(s,c~,s~,n,x) =      if conditionp(gname(c), s~)
                           then set_condition(s~, gname(c))
                      else mark_state_indeterminate(s~)
```

```
<similar_fields, s> ::= <identifier_list, is> COLON <type_specification, ft>
```

```
    field_descs(s,sn,ut,x) = let ftd = type_desc(ft,sn,ut,x)
                                 fns = record_field_names(is)
                                  if error_descp(ftd) then ftd
                             else pair_list_map
                                    (fns, ncopies(length(fns), ftd))

     record_field_names(s) = record_field_names(is)
```

```
<similar_formal_data_parameters, d> ::=
        <opt_access_specification, a> <identifier_list, is> COLON
        <type_specification, ft>
```

```
          access(d) = access(a)
```

```
     dparam_name(d) = dparam_name(is)

    formal_dargs(d) = full_dargs(access(a), is, ft)

     formal_type(d) = ft



<specification_expression, se> ::=
        <non_validated_specification_expression, se2>


    expression_from_spec(se) = expression_from_spec(se2)

              gp(se,c,s,n,x) = gp_update_assert
                                 (expression_from_spec(se), c, s, n, x)



<specification_expression, se> ::= <validated_specification_expression, se2>


    expression_from_spec(se) = expression_from_spec(se2)

              gp(se,c,s,n,x) = gp(se2, c, s, n - 1, x)



<specification_expression, se> ::=
        OPEN_PAREN <validated_specification_expression, se2> CLOSE_PAREN


    expression_from_spec(se) = expression_from_spec(se2)

              gp(se,c,s,n,x) = gp(se2, c, s, n - 1, x)



<statement, s> ::= <assert_specification, s2>


    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<statement, s> ::= <procedural_statement, s2>


    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<statement, s> ::= <procedure_composition_rule, s2>


    gp(s,c,s~,n,x) = gp(s2, c, s~, n - 1, x)



<statement_list, ss> ::= <statement, s>
```

```
    gp(ss,c,s~,n,x) = gp(s, c, s~, n - 1, x)
```

```
<statement_list, ss> ::= <statement_list, ss2> SEMI_COLON <statement, s>

    gp(ss,c,s~,n,x) = gp(s, c, gp(ss2, c, s~, n - 1, x), n - 1, x)
```

```
<type_declaration, d> ::= TYPE <IDENTIFIER, tn> EQUAL <type_definition, d2>

                  kind(d) = 'type

    pending_type_defnp(d) = pending_type_defnp(d2)

     scalar_type_defnp(d) = scalar_type_defnp(d2)

     scalar_value_list(d) = scalar_value_list(d2)

     type_desc(d,sn,ut,x) =      if pending_type_defnp(d)
                                 then pending_desc(unit_name(d), sn)
                            else if scalar_type_defnp(d)
                                 then construct_scalar_desc
                                        (unit_name(d), sn,
                                         scalar_value_list(d), x)
                            else type_desc(d2,sn,ut,x)

             unit_list(d) = cons(d, derived_units(tn,d2))

             unit_name(d) = unit_name(tn)
```

```
<type_definition, d> ::= PENDING

    pending_type_defnp(d) = t
```

```
<type_definition, d> ::= <array_type, a>

    type_desc(d,sn,ut,x) = type_desc(a,sn,ut,x)
```

```
<type_definition, d> ::= <mapping_type, m>

    type_desc(d,sn,ut,x) = type_desc(m,sn,ut,x)
```

```
<type_definition, d> ::= <record_type, r>

    type_desc(d,sn,ut,x) = type_desc(r,sn,ut,x)
```

```
<type_definition, d> ::= <scalar_type, s>

      derived_units(n,d) = derived_units(n,s)

    scalar_type_defnp(d) = t

    scalar_value_list(d) = scalar_value_list(s)



<type_definition, d> ::= <sequence_type, s>

    type_desc(d,sn,ut,x) = type_desc(s,sn,ut,x)



<type_definition, d> ::= <set_type, s>

    type_desc(d,sn,ut,x) = type_desc(s,sn,ut,x)



<type_definition, s> ::=
        <type_specification, s> <opt_default_initial_value_expression, v>

    type_desc(s,sn,ut,x) = set_default_value
                              (type_desc(s,sn,ut,x),
                               default_initial_value(v,sn,x))



<type_specification, s> ::= <IDENTIFIER, tn>

    type_desc(s,sn,ut,x) = type_desc(tn,sn,ut,x)



<type_specification, s> ::= <IDENTIFIER, tn> <range, r>

    type_desc(s,sn,ut,x) = set_range
                              (type_desc(tn,sn,ut,x), range_min(r,sn,x),
                               range_max(r,sn,x))



<unary_operator, op> ::= MINUS

            apply_unary_op(op,v) = gminus(v)

    gpf_apply_unary_op(op,sv,s0) = gpf_gminus(sv,s0)



<unary_operator, op> ::= NOT
```

```
          apply_unary_op(op,v) = gnot(v)

    gpf_apply_unary_op(op,sv,s0) = gpf_gnot(sv,s0)



<unit_declaration, d> ::= <constant_declaration, d2>

          kind(d) = kind(d2)

    local_name(d) = unit_name(d2)

     unit_list(d) = unit_list(d2)



<unit_declaration, d> ::= <function_declaration, d2>

          kind(d) = kind(d2)

    local_name(d) = unit_name(d2)

     unit_list(d) = unit_list(d2)



<unit_declaration, d> ::= <lemma_declaration, d2>

          kind(d) = kind(d2)

    local_name(d) = unit_name(d2)

     unit_list(d) = unit_list(d2)



<unit_declaration, d> ::= <procedure_declaration, d2>

          kind(d) = kind(d2)

    local_name(d) = unit_name(d2)

     unit_list(d) = unit_list(d2)



<unit_declaration, d> ::= <type_declaration, d2>

          kind(d) = kind(d2)

    local_name(d) = unit_name(d2)

     unit_list(d) = unit_list(d2)



<unit_or_name_declaration, d> ::= <name_declaration, d2>
```

```
    unit_list(d) = unit_list(d2)



<unit_or_name_declaration, d> ::= <unit_declaration, d2>


    unit_list(d) = unit_list(d2)



<unit_or_name_declaration_list, us> ::= <unit_or_name_declaration, d>


    unit_list(us) = unit_list(d)



<unit_or_name_declaration_list, us> ::=
        <unit_or_name_declaration_list, us2> SEMI_COLON
        <unit_or_name_declaration, d>


    unit_list(us) = unit_list(us2) @ unit_list(d)



<validated_specification_expression, se> ::=
        <non_validated_specification_expression, se2> OTHERWISE
        <IDENTIFIER, i>


    expression_from_spec(se) = expression_from_spec(se2)

              gp(se,c,s,n,x) = let r = gpf_type_check
                                         (boolean_desc,
                                          gpf
                                            (expression_from_spec(se), c, s,
                                             n - 1, x))
                                    if normal_state(r)
                                    then      if gtruep(result~(r))
                                              then gp_record_assert
                                                     (result~(r), s)
                                         else gp
                                                (mk_signal_stmt(i), c,
                                                 gp_record_assert
                                                   (result~(r), s), n - 1, x)
                               else r



<value_alterations, a> ::=
        WITH OPEN_PAREN <component_alterations_list, al> CLOSE_PAREN


      gf_modifiers(bv,a,c,v,n,x) = gf_modifiers(bv,al,c,v,n,x)

    gpf_modifiers(sbv,a,c,s,n,x) = gpf_modifiers(sbv, al, c, s, n - 1, x)



<value_list, v> ::= <expression, e>
```

```
                actual_dargs(v) = rcons(nil,e)

             gf_adp(v,c,v~,n,x) = rcons(nil, gf(e,c,v~,n,x))

    gf_element_list(v,c,v~,n,x) = rcons(nil, gf(e,c,v~,n,x))

    gf_element_type(v,c,v~,n,x) = base_type(type(gf(e,c,v~,n,x)))

             gpf_adp(v,c,s,n,x) = rcons(nil, gpf(e, c, s, n - 1, x))

    gpf_element_list(v,c,s,n,x) = rcons(nil, gpf(e, c, s, n - 1, x))

    gpf_element_type(v,c,s,n,x) = base_type
                                    (type(result~(gpf(e, c, s, n - 1, x))))



<value_list, v> ::= <value_list, v2> COMMA <expression, e>


                actual_dargs(v) = rcons(actual_dargs(v2), e)

             gf_adp(v,c,v~,n,x) = rcons(gf_adp(v2,c,v~,n,x), gf(e,c,v~,n,x))

    gf_element_list(v,c,v~,n,x) = rcons
                                    (gf_element_list(v2,c,v~,n,x),
                                     gf(e,c,v~,n,x))

    gf_element_type(v,c,v~,n,x) = gf_element_type(v2,c,v~,n,x)

             gpf_adp(v,c,s,n,x) = rcons
                                    (gpf_adp(v2, c, s, n - 1, x),
                                     gpf(e, c, s, n - 1, x))

    gpf_element_list(v,c,s,n,x) = rcons
                                    (gpf_element_list(v2, c, s, n - 1, x),
                                     gpf(e, c, s, n - 1, x))

    gpf_element_type(v,c,s,n,x) = gpf_element_type(v2, c, s, n - 1, x)



<value_modifiers, m> ::= <component_selectors, s>


                    arg_list(m) = arg_list(s)

                   arg_listp(m) = arg_listp(s)

         component_selectors(m) = s

     gf_modifiers(bv,m,c,v,n,x) = gf_modifiers(bv,s,c,v,n,x)

  gpf_modifiers(sbv,m,c,s~,n,x) = gpf_modifiers(sbv, s, c, s~, n - 1, x)



<value_modifiers, m> ::= <range, r>


     gf_modifiers(bv,m,c,v,n,x) = gf_modifiers(bv,r,c,v,n,x)

   gpf_modifiers(sbv,m,c,s,n,x) = gpf_modifiers(sbv, r, c, s, n - 1, x)
```

```
<value_modifiers, m> ::= <value_alterations, a>

      gf_modifiers(bv,m,c,v,n,x) = gf_modifiers(bv,a,c,v,n,x)

    gpf_modifiers(sbv,m,c,s,n,x) = gpf_modifiers(sbv, a, c, s, n - 1, x)
```

Appendix D
Meta-Functions

D.1 Functions Defining F

The function **F** referred to in the chapters of this report is the function **meta_F**.

```
; *************************************
;  Functions on Integers and Rationals
; *************************************

; integer and rational libraries
(note-lib "r2")

(defn ipower (x e)
  ; entry: (and (integerp x) (numberp e))
  (if (zerop e)
      1
    (itimes x (ipower x (sub1 e)))))

(defn rpower (x e)
  ; entry: (and (rationalp x) (numberp e))
  (if (zerop e)
      (rational 1 1)
    (rtimes x (rpower x (sub1 e)))))

(defn rleq (x y)
  (or (rlessp x y)
      (requal x y)))



;; *****************
;;  ASCII characters
;; *****************

(defn ascii_NUL () 0)
(defn ascii_SOH () 1)
(defn ascii_STX () 2)
(defn ascii_ETX () 3)
(defn ascii_EOT () 4)
(defn ascii_ENQ () 5)
(defn ascii_ACK () 6)
(defn ascii_BEL () 7)
(defn ascii_BS () 8)
(defn ascii_HT () 9)
(defn ascii_LF () 10)
(defn ascii_VT () 11)
(defn ascii_FF () 12)
(defn ascii_CR () 13)
(defn ascii_SO () 14)
(defn ascii_SI () 15)
(defn ascii_DLE () 16)
(defn ascii_DC1 () 17)
(defn ascii_DC2 () 18)
(defn ascii_DC3 () 19)
(defn ascii_DC4 () 20)
(defn ascii_NAK () 21)
(defn ascii_SYN () 22)
(defn ascii_ETB () 23)
(defn ascii_CAN () 24)
(defn ascii_EM () 25)
(defn ascii_SUB () 26)
(defn ascii_ESC () 27)
```

```
(defn ascii_FS () 28)
(defn ascii_GS () 29)
(defn ascii_RS () 30)
(defn ascii_US () 31)
(defn ascii_space () 32)
(defn ascii_exclamation_point () 33)
(defn ascii_double_quote () 34)
(defn ascii_number_sign () 35)
(defn ascii_dollar () 36)
(defn ascii_percent () 37)
(defn ascii_and () 38)
(defn ascii_single_quote () 39)
(defn ascii_open_paren () 40)
(defn ascii_close_paren () 41)
(defn ascii_star () 42)
(defn ascii_plus () 43)
(defn ascii_comma () 44)
(defn ascii_dash () 45)
(defn ascii_dot () 46)
(defn ascii_slash () 47)
(defn ascii_0 () 48)
(defn ascii_1 () 49)
(defn ascii_2 () 50)
(defn ascii_3 () 51)
(defn ascii_4 () 52)
(defn ascii_5 () 53)
(defn ascii_6 () 54)
(defn ascii_7 () 55)
(defn ascii_8 () 56)
(defn ascii_9 () 57)
(defn ascii_colon () 58)
(defn ascii_semicolon () 59)
(defn ascii_lt () 60)
(defn ascii_equal () 61)
(defn ascii_gt () 62)
(defn ascii_question () 63)
(defn ascii_at () 64)
(defn ascii_A () 65)
(defn ascii_B () 66)
(defn ascii_C () 67)
(defn ascii_D () 68)
(defn ascii_E () 69)
(defn ascii_F () 70)
(defn ascii_G () 71)
(defn ascii_H () 72)
(defn ascii_I () 73)
(defn ascii_J () 74)
(defn ascii_K () 75)
(defn ascii_L () 76)
(defn ascii_M () 77)
(defn ascii_N () 78)
(defn ascii_O () 79)
(defn ascii_P () 80)
(defn ascii_Q () 81)
(defn ascii_R () 82)
(defn ascii_S () 83)
(defn ascii_T () 84)
(defn ascii_U () 85)
(defn ascii_V () 86)
(defn ascii_W () 87)
(defn ascii_X () 88)
(defn ascii_Y () 89)
(defn ascii_Z () 90)
(defn ascii_open_bracket () 91)
(defn ascii_backslash () 92)
(defn ascii_close_bracket () 93)
(defn ascii_caret () 94)
(defn ascii_underscore () 95)
(defn ascii_back_quote () 96)
```

```
(defn ascii_lc_a () 97)
(defn ascii_lc_b () 98)
(defn ascii_lc_c () 99)
(defn ascii_lc_d () 100)
(defn ascii_lc_e () 101)
(defn ascii_lc_f () 102)
(defn ascii_lc_g () 103)
(defn ascii_lc_h () 104)
(defn ascii_lc_i () 105)
(defn ascii_lc_j () 106)
(defn ascii_lc_k () 107)
(defn ascii_lc_l () 108)
(defn ascii_lc_m () 109)
(defn ascii_lc_n () 110)
(defn ascii_lc_o () 111)
(defn ascii_lc_p () 112)
(defn ascii_lc_q () 113)
(defn ascii_lc_r () 114)
(defn ascii_lc_s () 115)
(defn ascii_lc_t () 116)
(defn ascii_lc_u () 117)
(defn ascii_lc_v () 118)
(defn ascii_lc_w () 119)
(defn ascii_lc_x () 120)
(defn ascii_lc_y () 121)
(defn ascii_lc_z () 122)
(defn ascii_open_brace () 123)
(defn ascii_vertical_bar () 124)
(defn ascii_close_brace () 125)
(defn ascii_tilde () 126)
(defn ascii_DEL () 127)


;; ***********
;;  Utilities
;; ***********

; --------------------------------
;  ASCII character (list) utilities
; --------------------------------

(defn ascii_characterp (c)
  (and (numberp c)
       (leq 0 c)
       (leq c 127)))

(defn ascii_character_listp (x)
  (if (nlistp x)
      (equal x 0)
    (and (ascii_characterp (car x))
         (ascii_character_listp (cdr x)))))

(defn is_digit (x)
  (and (numberp x)
       (leq (ascii_0) x)
       (leq x (ascii_9))))

(defn is_letter (x)
  (and (numberp x)
       (or (and (leq (ascii_A) x) (leq x (ascii_Z)))
           (and (leq (ascii_lc_a) x) (leq x (ascii_lc_z))))))

(defn printable_char_ordp (x)
  (and (numberp x)
       (leq (ascii_space) x)
       (leq x (ascii_tilde))))

(defn upper_case (c)
```

```
  (if (and (leq (ascii_lc_a) c) (leq c (ascii_lc_z)))
      (difference c 32)
      c))

(defn uc_list (u)
  (if (nlistp u)
      u
    (cons (upper_case (car u))
          (uc_list (cdr u)))))



; -----------------
;  Number Utilities
; -----------------

(defn number_list2 (hi lo nl)
  (if (zerop hi)
      (cons lo nl)
    (number_list2 (sub1 hi) lo (cons (iplus hi lo) nl))))

(defn number_list (lo hi)
  (if (ileq lo hi)
      (number_list2 (idifference hi lo) lo nil)
    nil))

(defn number_to_char_list (x)
  (if (numberp x)
      (let ((q (quotient x 10))
            (r (cons (plus (remainder x 10) (ascii_0)) 0)))
        (if (zerop q)
            r
          (append (number_to_char_list q) r)))
    nil))



; ---------------
;  List Utilities
; ---------------

(defn intersection (x y)
  (if (nlistp x)
      nil
    (if (member (car x) y)
        (cons (car x) (intersection (cdr x) y))
      (intersection (cdr x) y))))

(defn length (l)
  (if (nlistp l)
      (zero)
      (add1 (length (cdr l)))))

(defn ncopies (n x)
  (if (zerop n)
      nil
    (cons x (ncopies (sub1 n) x))))

(defn nth (i s)
  (if (nlistp s)
      nil
    (if (equal i 1)
        (car s)
      (nth (sub1 i) (cdr s)))))

(defn rcar (x)
  (if (nlistp x)
      nil
    (if (nlistp (cdr x))
```

```
        (car x)
      (rcar (cdr x)))))

(defn rcdr (x)
  (if (nlistp x)
      x
    (if (nlistp (cdr x))
        (cdr x)
      (cons (car x) (rcdr (cdr x))))))

(prove-lemma lessp_rcdr (rewrite)
  (implies (listp s)
           (lessp (count (rcdr s)) (count s))))

(defn rcons (x y)
  (if (nlistp x)
      (cons y x)
    (cons (car x) (rcons (cdr x) y))))

(prove-lemma rcar_rcons (rewrite)
  (equal (rcar (rcons x y)) y))

(prove-lemma rcdr_rcons (rewrite)
  (equal (rcdr (rcons x y)) x))

(defn remove (e s)
  (if (nlistp s)
      s
    (if (equal (car s) e)
        (remove e (cdr s))
      (cons (car s) (remove e (cdr s))))))

(prove-lemma lessp_remove_length (rewrite)
  (implies (member e s)
           (lessp (length (remove e s))
                  (length s)))
  ( (induct (remove e s)) ))

(defn set_difference (x y)
  (if (nlistp x)
      x
    (if (member (car x) y)
        (set_difference (cdr x) y)
      (cons (car x) (set_difference (cdr x) y)))))

(defn subsetp (x y)
  (if (nlistp x)
      T
    (and (member (car x) y)
         (subsetp (cdr x) y))))

(defn set_equal (x y)
  (and (subsetp x y)
       (subsetp y x)))


; ****************
;  Key-Value Maps
; ****************

(defn empty_map ()
  nil)

(defn map_entry (k v)
  (cons k v))

(defn in_map (m k)
  (assoc k m))
```

```
(defn add_to_map (m k v)
  (if (nlistp m)
      (list (map_entry k v))
    (if (equal (caar m) k)
        (cons (map_entry k v) (cdr m))
      (cons (car m) (add_to_map (cdr m) k v)))))

(defn mapped_value (m k)
  (let ((v (assoc k m)))
    (if (listp v)
        (cdr v)
      nil)))

(defn all_matches (k m)
  (if (nlistp m)
      nil
    (if (equal k (caar m))
        (cons (cdar m) (all_matches k (cdr m)))
      (all_matches k (cdr m)))))

(defn keys (x)
  (if (nlistp x)
      nil
    (cons (caar x) (keys (cdr x)))))

(defn key_values (x)
  (if (nlistp x)
      nil
    (cons (cdar x) (key_values (cdr x)))))

(defn key_value_mapp (m)
  (if (nlistp m)
      (equal m nil)
    (and (listp (car m))
         (key_value_mapp (cdr m)))))

(prove-lemma lessp_keys (rewrite)
  (implies (and (key_value_mapp m) (listp m))
           (lessp (count (keys m)) (count m))))

(prove-lemma lessp_key_values (rewrite)
  (implies (and (key_value_mapp m) (listp m))
           (lessp (count (key_values m)) (count m))))

(defn pair_list_map (x y)
  (if (nlistp x)
      nil
    (cons (map_entry (car x) (car y))
          (pair_list_map (cdr x) (cdr y)))))



; *******
;  Trees
; *******

(add-shell mk_tree nil treep
           ((root (none-of) false)
            (subtrees (none-of) false)))



; ****************************
;  Productions and Parse Trees
; ****************************

; A grammar symbol is a litatom, representing either a terminal or a
; nonterminal in the grammar.  A terminal grammar symbol is called a token.
```

```
; (prodn x y) is the representation of "x ::= y" and
; (lhs (prodn x y)) = x & (rhs (prodn x y)) = y.
;
; The left-hand-side (lhs) of a production (prodn) is a nonterminal grammar
; symbol.  The right-hand-side (rhs) of a production is either a grammar
; symbol or a list of grammar symbols.  The grammar symbols in a production
; may be tagged (see below).

(add-shell prodn nil prodnp
           ((lhs (none-of) false)
            (rhs (none-of) false)))


; (tag grammar_symbol the_tag) constructs a tagged grammar symbol.
; (gsymbol (tag grammar_symbol the_tag)) = grammar_symbol and
; (label (tag grammar_symbol the_tag)) = the_tag.
;
; Grammar symbols in productions are tagged so that semantic functions can be
; converted mechanically to report form, where the tags are thought to clarify
; the presentation.


(add-shell tag nil taggedp
           ((gsymbol (none-of) false)
            (label (none-of) false)))


; (mk_tree r s) is the representation of a parse tree and
; (root (mk_tree r s)) = r & (subtrees (mk_tree r s)) = s.
;
; The root of a parse tree is a grammar symbol, representing a terminal or a
; nonterminal.  Grammar symbols in parse trees are not tagged.  If the root
; represents a terminal, the parse tree is a leaf and its subtrees part is a
; representation of the lexeme that matched the terminal grammar symbol
; (token).  Otherwise, the root represents a nonterminal, and the subtrees
; part is a parse tree or a list of parse trees.



; ********************
;   Parse Tree Leaves
; ********************

; A leaf of a parse tree is a parse tree whose root is a terminal grammar
; symbol (token) and whose subtrees part is the list of characters (lexeme)
; that matched the token.  The last cdr of the character list should be zero,
; not nil.


(defn reserved_words ()
  '(ADJOIN ALL AND APPEND ARRAY ASSERT ASSUME AWAIT BEFORE BEGIN
    BEHIND BINARY BLOCK BUFFER CASE CBLOCK CENTRY CEXIT COBEGIN COND
    CONST DECIMAL DIFFERENCE DIV EACH ELEMENT ELIF ELSE END ENTRY EQ
    EXIT EXTENDS FI FROM FUNCTION GE GIVE GT HEX HOLD IF IFF IMP
    INPUT IN INTO INITIALLY INTERSECT IS KEEP LE LEAVE LEMMA LOOP LT
    MAPOMIT MAPPING MOD MOVE NAME NE NEW NORMAL NOT OCTAL OF OMIT
    ON OR OTHERWISE OUTPUT PENDING PROCEDURE PROVE RECEIVE RECORD
    REMOVE SCOPE SEND SEQ SEQOMIT SEQUENCE SET SIGNAL SOME SUB THEN
    TO TYPE UNION UNLESS VAR WHEN WITH

    ALIAS EXPORT IMPORT MULTIPLECOND NONE SPACE STRING VALUE))

(defn special_symbol_map ()
  (list ;   token        lexeme
        ;   -----        ------
    (cons   'AND         (cons (ascii_and) 0))
    (cons   'APPEND      (cons (ascii_at) 0))
    (cons   'CLOSE_PAREN (cons (ascii_close_paren) 0))
    (cons   'COLON       (cons (ascii_colon) 0))
```

```
    (cons   'COLON_EQUAL (cons (ascii_colon) (cons (ascii_equal) 0)))
    (cons   'COLON_GT    (cons (ascii_colon) (cons (ascii_gt) 0)))
    (cons   'COMMA       (cons (ascii_comma) 0))
    (cons   'DOT         (cons (ascii_dot) 0))
    (cons   'DOT_DOT     (cons (ascii_dot) (cons (ascii_dot) 0)))
    (cons   'EQUAL       (cons (ascii_equal) 0))
    (cons   'GT          (cons (ascii_gt) 0))
    (cons   'IMP         (cons (ascii_dash) (cons (ascii_gt) 0)))
    (cons   'LT          (cons (ascii_lt) 0))
    (cons   'LT_COLON    (cons (ascii_lt) (cons (ascii_colon) 0)))
    (cons   'MINUS       (cons (ascii_dash) 0))
    (cons   'OPEN_PAREN  (cons (ascii_open_paren) 0))
    (cons   'PLUS        (cons (ascii_plus) 0))
    (cons   'SEMI_COLON  (cons (ascii_semicolon) 0))
    (cons   'SLASH       (cons (ascii_slash) 0))
    (cons   'STAR        (cons (ascii_star) 0))
    (cons   'STAR_STAR   (cons (ascii_star) (cons (ascii_star) 0)))))

(defn special_symbols ()
  ; Tokens for special symbols.
  (keys (special_symbol_map)))

(defn tokens ()
  (append '(DIGIT_LIST IDENTIFIER CHARACTER_VALUE STRING_VALUE ENTRY_VALUE)
          (union (reserved_words) (special_symbols))))

(defn tokenp (x)
  (member x (tokens)))

(defn leafp (x)
  (and (treep x)
       (tokenp (root x))
       (ascii_character_listp (subtrees x))))

(defn lexeme (x)
  (if (leafp x)
      (subtrees x)
    nil))



; ========================
;  Recognizers for Lexemes
; ========================

; ---------------------
;  Reserved Word Lexeme
; ---------------------

(defn reserved_word_lexemep (x)
  (member (pack (uc_list x)) (reserved_words)))


; ----------------------
;  Special Symbol Lexeme
; ----------------------

(defn special_symbol_lexemes ()
  (key_values (special_symbol_map)))

(defn special_symbol_lexemep (x)
  (member x (special_symbol_lexemes)))


; -----------------------
;  Character Value Lexeme
; -----------------------

(defn character_value_lexemep (x)
```

```
  (and (equal (length x) 3)
       (equal (car x) (ascii_single_quote))
       (printable_char_ordp (cadr x))
       (equal (rcar x) (ascii_single_quote))
       (equal (cdddr x) 0)))


; ------------------
;  Digit List Lexeme
; ------------------

(defn is_hexdigit (d)
  (or (is_digit d)
      (and (leq (ascii_A) (upper_case d))
           (leq (upper_case d) (ascii_F)))))

(defn is_hexdigit_list (x)
  (if (nlistp x)
      (equal x 0)
    (and (is_hexdigit (car x))
         (is_hexdigit_list (cdr x)))))

(defn digit_list_lexemep (x)
  (and (listp x)
       (is_digit (car x))
       (is_hexdigit_list (cdr x))))


; ------------------------------------
;  Entry Value and Identifier Lexemes
; ------------------------------------

(defn identifier_lexeme_form (x)
  (if (nlistp x)
      (equal x 0)
    (if (equal (car x) (ascii_underscore))
        (and (listp (cdr x))
             (not (equal (cadr x) (ascii_underscore)))
             (identifier_lexeme_form (cdr x)))
      (and (or (is_letter (car x))
               (is_digit (car x)))
           (identifier_lexeme_form (cdr x))))))

(defn identifier_lexemep (x)
  (if (listp x)
      (and (is_letter (car x))
           (identifier_lexeme_form x)
           (not (member (pack (uc_list x)) (reserved_words))))
    F))

(defn entry_value_lexemep (x)
  (and (listp x)
       (identifier_lexemep (rcdr x))
       (equal (rcar x) (ascii_single_quote))))


; --------------------
;  String Value Lexeme
; --------------------

(defn string_char_listp (s)
  (if (nlistp s)
      (equal s 0)
    (if (equal (car s) (ascii_double_quote))
        (and (listp (cdr s))
             (equal (cadr s) (ascii_double_quote))
             (string_char_listp (cddr s)))
      (and (ascii_characterp (car s))
           (string_char_listp (cdr s))))))
```

```
(defn string_value_lexemep (x)
  (and (geq (length x) 2)
       (equal (car x) (ascii_double_quote))
       (equal (rcar x) (ascii_double_quote))
       (string_char_listp (rcdr (cdr x)))))



; ==================================
;  Recognizers for Parse Tree Leaves
; ==================================

(defn reserved_wordp (x)
  (and (leafp x)
       (member (root x) (reserved_words))
       (equal (uc_list (lexeme x))
              (uc_list (unpack (root x))))))

(defn special_symbolp (x)
  (and (leafp x)
       (member (root x) (special_symbols))
       (equal (lexeme x)
              (mapped_value (root x) (special_symbol_map)))))

(defn character_valuep (x)
  (and (leafp x)
       (equal (root x) 'CHARACTER_VALUE)
       (character_value_lexemep (lexeme x))))

(defn digit_listp (x)
  (and (leafp x)
       (equal (root x) 'DIGIT_LIST)
       (digit_list_lexemep (lexeme x))))

(defn entry_valuep (x)
  (and (leafp x)
       (equal (root x) 'ENTRY_VALUE)
       (entry_value_lexemep (lexeme x))))

(defn identifierp (x)
  (and (leafp x)
       (equal (root x) 'IDENTIFIER)
       (identifier_lexemep (lexeme x))))

(defn string_valuep (x)
  (and (leafp x)
       (equal (root x) 'STRING_VALUE)
       (string_value_lexemep (lexeme x))))



; *********************************************
;  Relation between Parse Trees and Productions
; *********************************************

; ==================
;  The Gypsy Grammar
; ==================

(dcl Gypsy_grammar ())


; ===========================================
;  First Rule Used in Derivation of Parse Tree
; ===========================================

(defn mk_rhs (pt)
  (if (treep pt)
      (root pt)
```

```
    (if (listp pt)
        (cons (mk_rhs (car pt))
              (mk_rhs (cdr pt)))
      pt)))

(defn mk_rule (pt)
  (if (treep pt)
      (prodn (root pt)
             (mk_rhs (subtrees pt)))
    nil))


; ========================
;  Well-Formed Parse Trees
; ========================

(defn parse_tree_leafp (pt)
  (if (leafp pt)
      (case (root pt)
        (CHARACTER_VALUE (character_valuep pt))
        (DIGIT_LIST (digit_listp pt))
        (ENTRY_VALUE (entry_valuep pt))
        (IDENTIFIER (identifierp pt))
        (STRING_VALUE (string_valuep pt))
        (otherwise (or (reserved_wordp pt)
                       (special_symbolp pt))))
    F))


(disable tokenp)
(disable parse_tree_leafp)
(disable mk_rule)

(do-mutual '(

(defn parse_treep (pt)
  (if (treep pt)
      (if (tokenp (root pt))
          (parse_tree_leafp pt)
        (and (litatom (root pt))
             (member (mk_rule pt) (Gypsy_grammar))
             (if (treep (subtrees pt))
                 (parse_treep (subtrees pt))
               (parse_tree_listp (subtrees pt)))))
               ; & (correct_precedence_rule_application pt)
    F)
  ( (lessp (count pt)) ))

(defn parse_tree_listp (sts)
  (if (listp sts)
      (and (parse_treep (car sts))
           (parse_tree_listp (cdr sts)))
    (equal sts nil))
  ( (lessp (count sts)) ))

))

(enable tokenp)
(enable parse_tree_leafp)
(enable mk_rule)


(constrain pT_intro (rewrite)
  (implies (not (parse_treep (pT s nt)))
           (equal (pT s nt) nil))
  ((pT (lambda (s nt) nil))))


; ===============================
```

```
;  Parse Tree/Production Matching
; ================================

(defn untag (pr)
  (if (prodnp pr)
      (prodn (untag (lhs pr)) (untag (rhs pr)))
  (if (taggedp pr)
      (gsymbol pr)
  (if (listp pr)
      (cons (untag (car pr)) (untag (cdr pr)))
    pr))))

(defn rule (pt pr)
  ; (rule pt pr) is T iff
  ;                 1.  pt is a well-formed parse tree
  ;                 2.  the first rule applied in pt's derivation is
  ;                     production pr of the Gypsy grammar
  (and (parse_treep pt)
       (equal (mk_rule pt) (untag pr))
       (member (untag pr) (Gypsy_grammar))))



; ********************
;  Functions on trees
; ********************

; -----------
;  Tree Size
; -----------

(defn tree_size (x)
  (if (treep x)
      (add1 (tree_size (subtrees x)))
  (if (listp x)
      (add1 (plus (tree_size (car x))
                  (tree_size (cdr x))))
    0)))


; ---------------
;  Tree Equality
; ---------------

(defn leaf_equal (t1 t2)
  (if (and (leafp t1) (leafp t2))
      (and (equal (root t1) (root t2))
           (if (member (root t1) '(CHARACTER_VALUE STRING_VALUE))
               (equal (lexeme t1) (lexeme t2))
             (equal (uc_list (lexeme t1))
                    (uc_list (lexeme t2)))))
    F))

(defn tree_equal (t1 t2)
  (if (leafp t1)
      (leaf_equal t1 t2)
  (if (treep t1)
      (and (treep t2)
           (equal (root t1) (root t2))
           (tree_equal (subtrees t1) (subtrees t2)))
  (if (listp t1)
      (and (listp t2)
           (tree_equal (car t1) (car t2))
           (tree_equal (cdr t1) (cdr t2)))
    (equal t1 t2)))))


; ----------
;  Subtrees
```

```
; ----------

(defn list_subtree (x n i)
  (if (nlistp x)
      nil
    (if (and (treep (car x))
             (equal (root (car x)) n))
        (if (equal i 1)
            (car x)
          (list_subtree (cdr x) n (sub1 i)))
      (list_subtree (cdr x) n i))))

(defn subtree (x n)
  (if (treep x)
      (if (treep (subtrees x))
          (if (equal (root (subtrees x)) n)
              (subtrees x)
            nil)
        (list_subtree (subtrees x) n 1))
    nil))

(defn subtree_body (x n)
  (if (treep (subtree x n))
      (subtrees (subtree x n))
    nil))

(defn subtree_i (x n i)
  (if (treep x)
      (list_subtree (subtrees x) n i)
    nil))

(prove-lemma lessp_list_subtree_size (rewrite)
  (implies (not (equal (tree_size (list_subtree x n i)) 0))
           (lessp (tree_size (list_subtree x n i))
                  (tree_size x))))

(prove-lemma lessp_subtree_size (rewrite)
  (implies (treep x)
           (lessp (tree_size (subtree x n))
                  (tree_size x))))

(prove-lemma lessp_subtree_i_size (rewrite)
  (implies (treep x)
           (lessp (tree_size (subtree_i x n i))
                  (tree_size x))))

(prove-lemma tree_size_not_zero (rewrite)
  (implies (treep x)
           (not (equal (tree_size x) 0))))

(prove-lemma lessp_subtree_body_size (rewrite)
  (implies (treep x)
           (lessp (tree_size (subtree_body x n))
                  (tree_size x)))
  ( (disable subtree)
    (use (lessp_subtree_size)) ))

(prove-lemma rule_imp_treep (rewrite)
  (implies (rule u (prodn x y))
           (treep u)))

(prove-lemma rule_imp_lessp_subtree_size (rewrite)
  (implies (rule u (prodn x y))
           (lessp (tree_size (subtree u z))
                  (tree_size u)))
  ( (disable rule subtree tree_size) ))

(prove-lemma rule_imp_lessp_subtree_i_size (rewrite)
  (implies (rule u (prodn x y))
```

```
           (lessp (tree_size (subtree_i u z i))
                  (tree_size u)))
  ( (disable rule subtree_i tree_size) ))

(defn subtreep (t1 t2)
  ; Is t1 a subtree of t2?
  (if (tree_equal t1 t2)
      T
    (if (listp t2)
        (or (subtreep t1 (car t2))
            (subtreep t1 (cdr t2)))
      (if (leafp t2)
          F
        (if (treep t2)
            (subtreep t1 (subtrees t2))
          F)))))


; ------------------
;  Tree Substitution
; ------------------

(defn subst_tree (t1 t2 t3)
  ; substitute t1 for t2 in t3
  (if (tree_equal t2 t3)
      t1
    (if (listp t3)
        (cons (subst_tree t1 t2 (car t3))
              (subst_tree t1 t2 (cdr t3)))
      (if (leafp t3)
          t3
        (if (treep t3)
            (mk_tree (root t3)
                     (subst_tree t1 t2 (subtrees t3)))
          t3)))))



; ******************
;  Tree Constructors
; ******************

; --------
;  Leaves
; --------

(defn mk_reserved_word (k)
  (if (member k (reserved_words))
      (mk_tree k (unpack k))
    nil))

(defn special_symbol_lexeme (x)
  (if (in_map (special_symbol_map) x)
      (mapped_value (special_symbol_map) x)
    nil))

(defn mk_special_symbol (x)
  (let ((r (special_symbol_lexeme x)))
    (if (listp r)
        (mk_tree x r)
      nil)))

(defn mk_unary_operator (x)
  (if (member x (reserved_words))
      (mk_tree 'unary_operator (mk_reserved_word x))
    (mk_tree 'unary_operator (mk_special_symbol x))))

(defn mk_digit_list (e)
  (if (digit_listp e)
```

```
      e
  (if (numberp e)
      (mk_tree 'DIGIT_LIST (number_to_char_list e))
    (mk_tree 'DIGIT_LIST e)))) ; Check e?

(defn mk_identifier_lexeme (id)
  (let ((n (if (litatom id) (unpack id) id)))
    (if (equal n (unpack 'NIL~))
        (unpack nil)
    (if (identifier_lexemep n)
        n
      nil))))

(defn mk_identifier (id)
  (if (equal (root id) 'IDENTIFIER)
      id
    (let ((n (mk_identifier_lexeme id)))
      (if (identifier_lexemep n)
          (mk_tree 'IDENTIFIER n)
        nil))))

(defn mk_entry_value_lexeme (id)
  (let ((n1 (if (litatom id) (unpack id) id)))
  (let ((n2 (if (equal (rcar n1) (ascii_single_quote))
                (mk_identifier_lexeme (rcdr n1))
              (mk_identifier_lexeme n1))))
    (if (identifier_lexemep n2)
        (rcons n2 (ascii_single_quote))
      nil))))

(defn mk_entry_value (id)
  (if (equal (root id) 'ENTRY_VALUE)
      id
  (if (equal (root id) 'IDENTIFIER)
      (mk_tree 'ENTRY_VALUE
               (rcons (lexeme id) (ascii_single_quote)))
    (let ((n (mk_entry_value_lexeme id)))
      (if (entry_value_lexemep n)
          (mk_tree 'ENTRY_VALUE n)
        nil)))))


; need:  CHARACTER_VALUE STRING_VALUE


; ------------
;  Non-Leaves
; ------------

(defn mk_empty ()
  (mk_tree 'empty nil))

; Some (all?) of the following need work.

(defn mk_number (e) ; ?
  (if (numberp e)
      (mk_tree 'number (mk_digit_list e))
    (mk_tree 'number e)))

(defn mk_literal_value (e) ; ?
  (if (numberp e)
      (mk_tree 'literal_value (mk_number e))
    (mk_tree 'literal_value e)))

(defn mk_primary_value (e)
  (if (numberp e)
      (mk_tree 'primary_value (mk_literal_value e))
  (if (litatom e)
      (mk_tree 'primary_value (mk_identifier e))
```

```
    (mk_tree 'primary_value e))))

(defn mk_modified_primary_value (e)
  (mk_tree 'modified_primary_value
           (mk_primary_value e)))

(defn mk_expression (e)
  (if (or (numberp e) (litatom e))
      (mk_tree 'expression (mk_modified_primary_value e))
    (mk_tree 'expression e)))

(defn mk_scalar_const_unit (n v q)
  ;; n is an identifier, v is a gname, q is a numberp.
  ;; <constant declaration> ::=
  ;;    CONST <IDENTIFIER> : <type specification> := <constant body>
  (mk_tree 'constant_declaration
           (list (mk_reserved_word 'CONST)
                 (mk_identifier v)
                 (mk_special_symbol 'COLON)
                 (mk_tree 'type_specification n)
                 (mk_special_symbol 'COLON_EQUAL)
                 (mk_tree 'constant_body
                          (mk_expression q)))))

(defn mk_named_unit (u s)
  ; u is local_renaming, s is an IDENTIFIER (the foreign_scope_name)
  (mk_tree 'name_declaration
           (list (mk_reserved_word 'NAME) (mk_tree 'local_aliases u)
                 (mk_reserved_word 'FROM) s)))

(defn mk_single_formal_data_parameter (a p ft)
  ; a is 'CONST or 'VAR, p is an identifier, ft is a type_specification.
  (mk_tree 'similar_formal_data_parameters
           (list (mk_tree 'opt_access_specification (mk_reserved_word a))
                 (mk_tree 'identifier_list p)
                 (mk_special_symbol 'COLON)
                 ft)))

(defn mk_arg_list (x)
  (mk_tree 'arg_list
           (list (mk_special_symbol 'OPEN_PAREN) x
                 (mk_special_symbol 'CLOSE_PAREN))))

(defn mk_actual (n)
  (mk_tree 'expression
           (mk_tree 'modified_primary_value
                    (mk_tree 'primary_value
                             (mk_identifier n)))))

(defn mk_actual_list (as n)
  (if (equal as nil)
      (mk_tree 'value_list (mk_actual n))
    (mk_tree 'value_list
             (list as (mk_special_symbol 'COMMA)
                   (mk_actual n)))))

(defn namelist_to_actuals (ns as)
  (if (nlistp ns)
      (if (equal as nil)
          as
        (mk_arg_list as))
    (namelist_to_actuals (cdr ns)
                         (mk_actual_list as (car ns)))))

(defn mk_component_selectors (x)
  (mk_tree 'component_selectors x))

(defn mk_value_modifiers (x)
  (if (equal (root x) 'arg_list)
```

```
      (mk_tree 'value_modifiers
               (mk_component_selectors x))
    (mk_tree 'value_modifiers x)))

(defn mk_true_expression ()
  (mk_tree 'expression
           (mk_tree 'modified_primary_value
                    (mk_primary_value 'true))))

(defn mk_identifier_list (is i)
  (if (equal is nil)
      (mk_tree 'identifier_list i)
    (mk_tree 'identifier_list
             (list is (mk_special_symbol 'COMMA) i))))

(defn mk_bound_expression (qn ts be)
  (mk_tree 'bound_expression
           (list qn (mk_special_symbol 'COLON)
                 ts (mk_special_symbol 'COMMA) be)))

(defn mk_quantified_expression (qf qn ts be)
  (mk_tree 'expression
           (list (if (litatom qf) (mk_reserved_word qf) qf)
                 (mk_bound_expression qn ts be))))



;; ********
;;  errors
;; ********

(defn mk_error (y)
  (mk_tree 'error* y))

(defn mk_error_decl (y)
  (mk_error y))

(defn errorp (x)
  (and (treep x) (equal (root x) 'error*)))

(defn error_msg (x)
  (if (errorp x)
      (subtrees x)
    '(no error message)))

(defn actual_formal_type_error (a ft)
  (mk_error (list 'Actual 'parameter a 'is 'not 'in 'formal 'type ft)))

(defn adjoin_args_error (v)
  (mk_error (list 'Cannot 'ADJOIN 'to v 'because 'it 'is 'not 'a 'set)))

(defn alias_id_error (i sn)
  (mk_error_decl (cons sn (append '(is not the home scope of unit)
                                  (list i)))))

(defn append_args_error (v1 v2)
  (mk_error (list 'APPEND 'is 'not 'defined 'on v1 'and v2 'because 'they 'are
                  'not 'sequences)))

(defn array_index_error (id)
  (mk_error (list 'array 'index 'type id 'is 'not 'a 'nonrational 'simple
                  'type)))

(defn bad_string_error (s)
  (mk_error (list s 'is 'not 'a 'well_formed 'string)))

(defn bad_value_modifiers_error (x)
  (mk_error (list 'ill_formed 'value_modifiers x)))
```

```
(defn character_error (x)
  (mk_error (list x ’is ’not ’a ’character)))

(defn colon_gt_args_error ()
  (mk_error ’(Can colon_gt only to sequences)))

(defn component_assign_error (x)
  (mk_error (list ’Components ’of ’value x ’cannot ’be ’assigned)))

(defn condition_params_error (e)
  (mk_error (list ’Expression e ’should ’not ’have ’actual ’condition
                  ’parameters)))

(defn difference_args_error (v1 v2)
  (mk_error (list ’DIFFERENCE ’is ’not ’defined ’on v1 ’and v2 ’because ’they
                  ’are ’neither ’mappings ’nor ’sets)))

(defn domain_arg_error ()
  (mk_error ’(DOMAIN is defined only on mappings)))

(defn duplicate_field_names_error (d)
  (mk_error (list d ’has ’duplicate ’field ’names)))

(defn duplicate_param_names_error (n)
  (mk_error (list ’Duplicate ’formal ’parameter ’names n)))

(defn each_id_type_error (e c)
  (mk_error (list ’The ’bound ’identifier ’type ’in e ’of ’scope c ’is ’not
                  ’a ’bounded ’index ’type)))

(defn empty_seq_error (fn s)
  (mk_error (list fn ’of ’an ’empty ’sequence s)))

(defn empty_type_error (td)
  (mk_error (list ’Type td ’is ’empty)))

(defn entry_not_true_error (fn sn)
  (mk_error (list ’The ’entry ’spec ’is ’not ’true
                  ’for ’function fn ’in ’scope sn)))

(defn field_name_reserved_error (fds)
  (mk_error (list ’Reserved ’identifier ’cannot ’be ’used ’as
                  ’a ’record ’field ’name ’in fds)))

(defn first_arg_error ()
  (mk_error ’(FIRST is defined only on sequences)))

(defn function_access_error (fp)
  (mk_error (list ’Function ’formal ’parameter ’cannot ’be ’var fp)))

(defn if_test_not_boolean_error (e sn)
  (mk_error (list ’The ’if ’test ’of ’expression e ’in ’scope sn ’is ’not ’a
                  ’boolean ’valued ’expression)))

(defn in_arg_error (v)
  (mk_error (list ’IN ’is ’not ’defined ’on v ’because ’it ’is ’neither ’a
                  ’sequence ’nor ’a ’set)))

(defn indeterminate_fn_result_error (fn sn)
  (mk_error (list ’Function fn ’of ’scope sn ’returned
                  ’an ’indeterminate ’value)))

(defn intersect_args_error (v1 v2)
  (mk_error (list ’INTERSECT ’is ’not ’defined ’on v1 ’and v2 ’because ’they
                  ’are ’neither ’mappings ’nor ’sets)))

(defn last_arg_error ()
  (mk_error ’(LAST is defined only on sequences)))
```

```
(defn lower_pred_error (td)
  (mk_error (list 'PRED 'of 'lower 'of 'type td)))

(defn lt_colon_args_error ()
  (mk_error '(Can lt_colon only to sequences)))

(defn many_post_conditions_error (u c)
  (mk_error (list u 'has 'more 'than 'one 'exit 'specification
                 'for 'condition c)))

(defn many_scope_error (sn)
  (mk_error_decl (append '(There are several scopes named)
                        (list sn))))

(defn many_unit_error (i sn)
  (mk_error_decl (cons 'Scope
                       (cons sn
                             (append '(has several units named)
                                     (list i))))))

(defn mapping_merge_error (v1 v2)
  (mk_error (list 'Mappings v1 'and v2 'have 'elements 'with 'the 'same
                 'selector 'and 'different 'component 'values)))

(defn mapping_selector_type_error (sd)
  (mk_error (list 'Mapping 'selector 'type sd 'is 'not 'an 'equality 'type)))

(defn max_arg_error ()
  (mk_error '(MAX is defined only on integers and rationals)))

(defn min_arg_error ()
  (mk_error '(MIN is defined only on integers and rationals)))

(defn n_too_small ()
  (mk_error (list 'n 'is 'too 'small)))

(defn name_already_in_use_error (n)
  (mk_error (list 'The 'name n 'is 'already 'in 'use)))

(defn negative_exponent_error ()
  (mk_error '(Negative exponent)))

(defn no_function_defn_error (fn sn)
  (mk_error (list fn 'in 'scope sn 'has 'no 'definition)))

(defn no_scope_error (sn)
  (mk_error_decl (append '(There is no scope) (list sn))))

(defn no_such_component_error (s i)
  (mk_error (list s 'has 'no 'component i)))

(defn no_such_field_error (r n)
  (mk_error (list r 'has 'no 'field n)))

(defn no_unit_error (i sn)
  (mk_error_decl (list 'Unit i 'is 'not 'in 'scope sn)))

(defn non_simple_subrange_type_error (td)
  (mk_error (list 'Subrange 'of 'non-simple 'type td)))

(defn nonfirst_arg_error ()
  (mk_error '(NONFIRST is defined only on sequences)))

(defn nonlast_arg_error ()
  (mk_error '(NONLAST is defined only on sequences)))

(defn not_array_error (x)
  (mk_error (list x 'is 'not 'an 'array)))
```

```
(defn not_binary_op_error (op)
  (mk_error (list op 'is 'not 'a 'binary 'operator)))

(defn not_defined_on_type_error (op td)
  (mk_error (list op 'is 'not 'defined 'on 'type td)))

(defn not_equality_type_error (td)
  (mk_error (list td 'is 'not 'an 'equality 'type)))

(defn not_expression_error (e)
  (mk_error (list e 'is 'not 'an 'expression)))

(defn not_function_or_const_error (fn sn)
  (mk_error (list fn 'of 'scope sn 'is 'not 'a 'function 'or 'constant)))

(defn not_in_set_error (e s)
  (mk_error (list e 'is 'not 'in 'set s)))

(defn not_in_type_error (v td)
  (mk_error (list v 'is 'not 'in 'type td)))

(defn not_mapping_error (x)
  (mk_error (list x 'is 'not 'a 'mapping)))

(defn not_mapping_type_error (td)
  (mk_error (list td 'is 'not 'a 'mapping 'type)))

(defn not_range_error (r sn)
  (mk_error (list r 'of 'scope sn 'is 'not 'a 'range)))

(defn not_record_error (x)
  (mk_error (list x 'is 'not 'a 'record)))

(defn not_record_fields_error (s sn)
  (mk_error (list s 'of 'scope sn 'is 'not 'a 'parse 'tree 'for 'record 'field
                  'names)))

(defn not_selectable_error (x)
  (mk_error (list 'Components 'of x 'cannot 'be 'selected)))

(defn not_sequence_error (s)
  (mk_error (list s 'is 'not 'a 'sequence)))

(defn not_sequence_type_error (td)
  (mk_error (list td 'is 'not 'a 'sequence 'type)))

(defn not_set_type_error (td)
  (mk_error (list td 'is 'not 'a 'set 'type)))

(defn not_type_descriptor_error (td)
  (mk_error (list td 'is 'not 'a 'type 'descriptor)))

(defn not_type_error (tt sn)
  (mk_error (list tt 'of 'scope sn 'is 'not 'a 'type)))

(defn not_unary_op_error (op)
  (mk_error (list op 'is 'not 'a 'unary 'operator)))

(defn null_undefined_error (td)
  (mk_error (list 'NULL 'is 'not 'defined 'on 'type td)))

(defn number_error (d b)
  (mk_error (list d 'is 'not 'a 'number 'in 'base b)))

(defn omit_args_error (v)
  (mk_error (list 'Cannot 'OMIT 'from v 'because 'it 'is 'not 'a 'set)))

(defn opt_default_value_error (d sn)
  (mk_error (list d 'in 'scope sn 'is 'not 'an
```

```
                  'opt_default_initial_value_expression)))

(defn opt_size_limit_error (r sn)
  (mk_error (list r 'of 'scope sn 'is 'not 'an 'opt_size_limit_restriction)))

(defn ord_arg_error ()
  (mk_error '(ORD is defined only on scalar types)))

(defn param_reserved_error (n)
  (mk_error (list 'Reserved 'identifier n 'cannot 'be 'used 'as
                  'a 'formal 'parameter 'name)))

(defn pending_default_value_error (td sn)
  (mk_error (list 'Cannot 'compute 'default 'value 'for 'pending
                  'type td 'in 'scope sn)))

(defn pending_in_type_error (td)
  (mk_error (list 'In_type 'of 'pending 'type td)))

(defn pending_type_value_set_error (td)
  (mk_error (list 'Cannot 'compute 'value 'set 'of 'pending 'type td)))

(defn pred_arg_error ()
  (mk_error '(PRED is defined only on scalar types)))

(defn range_arg_error ()
  (mk_error '(RANGE is defined only on mappings)))

(defn range_limits_error (lo hi)
  (mk_error (list lo 'and hi 'are 'not 'of 'the 'same
                  'non_rational 'simple 'type)))

(defn rational_value_set_error (td)
  (mk_error (list 'Cannot 'compute 'the 'value 'set 'of 'rational 'type td)))

(defn scale_int_arg_error (i tn)
  (mk_error (list i 'cannot 'be 'SCALEd 'in 'type tn)))

(defn scale_type_arg_error (tn)
  (mk_error (list 'SCALE 'is 'not 'defined 'on 'type tn 'because
                  'it 'is 'not 'a 'scalar 'type)))

(defn scope_id_error (sn)
  (mk_error_decl (append '(Scope name)
                         (cons sn '(cannot be used as a unit name)))))

(defn scope_reserved_error (sn)
  (mk_error_decl (list 'Reserved 'identifier sn
                       'cannot 'be 'used 'as 'a 'scope 'name)))

(defn size_arg_error ()
  (mk_error '(SIZE is defined only on mappings sequences sets)))

(defn size_limit_error (d sn)
  (mk_error (list 'Size 'limit d 'of 'scope sn 'is 'not 'a 'non-negative
                  'integer 'expression)))

(defn sub_args_error (v1 v2)
  (mk_error (list 'SUB 'is 'not 'defined 'on v1 'and v2 'because 'they 'are
                  'not 'mappings 'or 'sequences 'or 'sets)))

(defn succ_arg_error ()
  (mk_error '(SUCC is defined only on scalar types)))

(defn type_defn_cycle_error (tn sn)
  (mk_error (list 'Cycle 'in 'type 'definition 'on tn 'in 'scope sn)))

(defn type_error (td sn)
  (mk_error (list 'Error 'in 'type td 'of 'scope sn)))
```

```
(defn unbounded_sequence_value_set_error ()
  (mk_error (list 'Cannot 'compute 'the 'value 'set 'of 'a 'sequence
                  'without 'a 'size 'limit 'restriction)))

(defn unbounded_type_error (td)
  (mk_error (list 'Type td 'is 'not 'bounded)))

(defn unbounded_value_set_error (td)
  (mk_error (list 'Cannot 'compute 'the 'value 'set 'of 'unbounded 'type td)))

(defn union_arg_error (v1 v2)
  (mk_error (list 'UNION 'defined 'only 'on 'mappings 'and 'sets v1 v2)))

(defn unit_reserved_error (u)
  (mk_error_decl (list 'Reserved 'identifier u
                       'cannot 'be 'used 'as 'a 'unit 'name)))

(defn unknown_name_error (n)
  (mk_error (list 'The 'name n 'is 'unknown)))

(defn upper_succ_error (td)
  (mk_error (list 'SUCC 'of 'upper 'of 'type td)))

(defn upper_undefined_error (td)
  (mk_error (list 'UPPER 'is 'not 'defined 'on 'type td)))

(defn zero_divide_error ()
  (mk_error '(Divide by zero)))

(defn zero_to_the_zero_power_error ()
  (mk_error '(Zero to the zero power)))



; **************************************
;  Semantic functions - tree extraction
; **************************************

(disable rule)
(disable subtree) (disable subtree_body) (disable subtree_i)


(defn gname (u)
  (if (identifierp u)
      (let ((n (uc_list (lexeme u))))
        (if (equal (pack n) nil)
            'NIL~   ; Instead of ~ can use $ ^ & * _ - + = < > ?
          (pack n)))
    nil))


(defn entry_name (e)
  (if (entry_valuep e)
      (let ((n (uc_list (lexeme e))))
        (if (equal (pack (rcdr n)) nil)
            (pack (rcons (rcons (rcdr n) (ascii_tilde)) (rcar n)))
          (pack n)))
    nil))


(defn access (a)

  (if (rule a (prodn (tag 'similar_formal_data_parameters 'd)
                     (list (tag 'opt_access_specification 'a)
                           (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ft))))
      (access (subtree a 'opt_access_specification))

  (if (rule a (prodn (tag 'internal_data_or_condition_objects 'iv)
```

```
                     (list (tag 'access_specification 'a)
                           (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ts)
                           (tag 'opt_internal_initial_value 'v)
                           'SEMI_COLON)))
      (access (subtree a 'access_specification))

  (if (rule a (prodn (tag 'opt_access_specification 'a)
                     'empty))
      'const

  (if (rule a (prodn (tag 'opt_access_specification 'a)
                     (tag 'access_specification 'a2)))
      (access (subtree a 'access_specification))

  (if (rule a (prodn (tag 'access_specification 'a)
                     'VAR))
      'var

  (if (rule a (prodn (tag 'access_specification 'a)
                     'CONST))
      'const

    nil))))))

  ( (lessp (tree_size a)) ))


(defn arg_list (e)

  (if (rule e (prodn (tag 'expression 'e)
                     (tag 'modified_primary_value 'm)))
      (arg_list (subtree e 'modified_primary_value))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'value_modifiers 'vm))))
      (arg_list (subtree e 'value_modifiers))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'actual_condition_parameters 'cp))))
      (arg_list (subtree e 'modified_primary_value))

  (if (rule e (prodn (tag 'value_modifiers 'm)
                     (tag 'component_selectors 's)))
      (arg_list (subtree e 'component_selectors))

  (if (rule e (prodn (tag 'component_selectors 's)
                     (tag 'arg_list 'as)))
      (subtree e 'arg_list)

  (if (rule e (prodn (tag 'arg_list 'as)
                     (list 'OPEN_PAREN (tag 'value_list 'vs)
                           'CLOSE_PAREN)))
      e

    nil))))))

  ( (lessp (tree_size e)) ))


(defn arg_listp (x)

  (if (rule x (prodn (tag 'value_modifiers 'm)
                     (tag 'component_selectors 's)))
      (arg_listp (subtree x 'component_selectors))

  (if (rule x (prodn (tag 'component_selectors 's)
```

```
                   (tag ’arg_list ’as)))
      T

  (if (rule x (prodn (tag ’arg_list ’as)
                   (list ’OPEN_PAREN (tag ’value_list ’vs)
                         ’CLOSE_PAREN)))
      T

    nil)))

  ( (lessp (tree_size x)) ))


(defn bound_boolean_expression (e)

  (if (rule e (prodn (tag ’expression ’e)
                   (list ’ALL (tag ’bound_expression ’b))))
      (bound_boolean_expression (subtree e ’bound_expression))

  (if (rule e (prodn (tag ’expression ’e)
                   (list ’SOME (tag ’bound_expression ’b))))
      (bound_boolean_expression (subtree e ’bound_expression))

  (if (rule e (prodn (tag ’bound_expression ’b)
                   (list (tag ’identifier_list ’q) ’COLON
                         (tag ’type_specification ’s) ’COMMA
                         (tag ’expression ’e))))
      (subtree e ’expression)

    nil)))

  ( (lessp (tree_size e)) ))


(defn bound_id (e)

  (if (rule e (prodn (tag ’expression ’e)
                   (list ’ALL (tag ’bound_expression ’b))))
      (bound_id (subtree e ’bound_expression))

  (if (rule e (prodn (tag ’expression ’e)
                   (list ’SOME (tag ’bound_expression ’b))))
      (bound_id (subtree e ’bound_expression))

  (if (rule e (prodn (tag ’bound_expression ’b)
                   (list (tag ’identifier_list ’q) ’COLON
                         (tag ’type_specification ’s) ’COMMA
                         (tag ’expression ’e))))
      ; The identifier_list holds the quantified_names.
      (bound_id (subtree e ’identifier_list))

  (if (rule e (prodn (tag ’identifier_list ’is)
                   (list (tag ’identifier_list ’is2) ’COMMA
                         (tag ’IDENTIFIER ’i))))
      (bound_id (subtree e ’identifier_list))

  (if (rule e (prodn (tag ’identifier_list ’is)
                   (tag ’IDENTIFIER ’i)))
      (bound_id (subtree e ’IDENTIFIER))

  (if (rule e (prodn (tag ’opt_each_clause ’e)
                   (list ’EACH (tag ’IDENTIFIER ’i) ’COLON
                         (tag ’type_specification ’ts) ’COMMA)))
      (bound_id (subtree e ’IDENTIFIER))

  (if (identifierp e)
      (gname e)

    nil)))))))
```

```
  ( (lessp (tree_size e)) ))


(defn bound_id_type (e)

  (if (rule e (prodn (tag 'expression 'e)
                     (list 'ALL (tag 'bound_expression 'b))))
      (bound_id_type (subtree e 'bound_expression))

  (if (rule e (prodn (tag 'expression 'e)
                     (list 'SOME (tag 'bound_expression 'b))))
      (bound_id_type (subtree e 'bound_expression))

  (if (rule e (prodn (tag 'bound_expression 'b)
                     (list (tag 'identifier_list 'q) 'COLON
                           (tag 'type_specification 's) 'COMMA
                           (tag 'expression 'e))))
      (subtree e 'type_specification)

  (if (rule e (prodn (tag 'opt_each_clause 'e)
                     (list 'EACH (tag 'IDENTIFIER 'i) 'COLON
                           (tag 'type_specification 'ts) 'COMMA)))
      (subtree e 'type_specification)

    nil))))

  ( (lessp (tree_size e)) ))


(defn case_exit_list2 (ls e c)

  (if (rule ls (prodn (tag 'case_exit_labels 'ls)
                      (list (tag 'case_exit_labels 'ls2) 'COMMA
                            (tag 'exit_label 'l))))
      (append (case_exit_list2 (subtree ls 'case_exit_labels) e c)
              (case_exit_list2 (subtree ls 'exit_label) e c))

  (if (rule ls (prodn (tag 'case_exit_labels 'ls)
                      (tag 'exit_label 'l)))
      (case_exit_list2 (subtree ls 'exit_label) e c)

  (if (rule ls (prodn (tag 'exit_label 'l)
                      (tag 'IDENTIFIER 'n)))
      (case_exit_list2 (subtree ls 'IDENTIFIER) e c)

  (if (rule ls (prodn (tag 'exit_label 'l)
                      'NORMAL))
      (if (equal c 'NORMAL)
          (list e)
        nil)

  (if (identifierp ls)
      (if (equal (gname ls) c)
          (list e)
        nil)

    nil)))))

  ( (lessp (tree_size ls)) ))


(defn case_exit_list (u c)

  (if (rule u (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (case_exit_list (subtree u 'procedure_body) c)
```

```
(if (rule u (prodn (tag 'function_declaration 'd)
                   (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                         (tag 'opt_external_data_objects 'a)
                         'COLON (tag 'type_specification 'rt)
                         (tag 'opt_external_conditions 'c)
                         'EQUAL (tag 'procedure_body 'b))))
    (case_exit_list (subtree u 'procedure_body) c)

(if (rule u (prodn (tag 'procedure_body 'b)
                   'PENDING))
    nil

(if (rule u (prodn (tag 'procedure_body 'b)
                   (list 'BEGIN
                         (tag 'external_operational_specification 'es)
                         (tag 'opt_internal_environment 'iv)
                         (tag 'opt_keep_specification 'k)
                         (tag 'opt_internal_statements 'st)
                         'END)))
    (case_exit_list (subtree u 'external_operational_specification) c)

(if (rule u (prodn (tag 'external_operational_specification 's)
                   (list (tag 'opt_entry_specification 'e)
                         (tag 'opt_exit_specification 'x))))
    (case_exit_list (subtree u 'opt_exit_specification) c)

(if (rule u (prodn (tag 'opt_exit_specification 'e)
                   'empty))
    nil

(if (rule u (prodn (tag 'opt_exit_specification 'e)
                   (list 'EXIT
                         (tag 'non_validated_specification_expression
                              'se)
                         'SEMI_COLON)))
    (if (equal c 'NORMAL)
        (list (subtree u 'non_validated_specification_expression))
      nil)

(if (rule u (prodn (tag 'opt_exit_specification 'e)
                   (list 'EXIT (tag 'conditional_exit_specification 'c)
                         'SEMI_COLON)))
    (case_exit_list (subtree u 'conditional_exit_specification) c)

(if (rule u (prodn (tag 'conditional_exit_specification 'c)
                   (list 'CASE 'OPEN_PAREN (tag 'case_exit_body 'e)
                         'CLOSE_PAREN)))
    (case_exit_list (subtree u 'case_exit_body) c)

(if (rule u (prodn (tag 'case_exit_body 'b)
                   (tag 'case_exit 'c)))
    (case_exit_list (subtree u 'case_exit) c)

(if (rule u (prodn (tag 'case_exit_body 'b)
                   (list (tag 'case_exit_body 'b2) 'SEMI_COLON
                         (tag 'case_exit 'c))))
    (append (case_exit_list (subtree u 'case_exit_body) c)
            (case_exit_list (subtree u 'case_exit) c))

(if (rule u (prodn (tag 'case_exit 'ce)
                   (list 'IS (tag 'case_exit_labels 'l) 'COLON
                         (tag 'non_validated_specification_expression
                              'e))))
    (case_exit_list2 (subtree u 'case_exit_labels)
                     (subtree u 'non_validated_specification_expression)
                     c)

  nil))))))))))))
```

```
  ( (lessp (tree_size u)) ))


(defn cdr_quantified_names (e)

  (if (or (rule e (prodn (tag 'expression 'e)
                         (list 'ALL (tag 'bound_expression 'b))))
          (rule e (prodn (tag 'expression 'e)
                         (list 'SOME (tag 'bound_expression 'b)))))
      (cdr_quantified_names (subtree e 'bound_expression))

  (if (rule e (prodn (tag 'bound_expression 'b)
                     (list (tag 'identifier_list 'q) 'COLON
                           (tag 'type_specification 's) 'COMMA
                           (tag 'expression 'e))))
      (cdr_quantified_names (subtree e 'identifier_list))

  (if (rule e (prodn (tag 'identifier_list 'is)
                     (list (tag 'identifier_list 'is2) 'COMMA
                           (tag 'IDENTIFIER 'i))))
      (mk_identifier_list (cdr_quantified_names
                            (subtree e 'identifier_list))
                          (subtree e 'IDENTIFIER))

    nil)))
  ( (lessp (tree_size e)) ))


(defn quantifier (e)
  ; Expand to include <each clause>?

  (if (rule e (prodn (tag 'expression 'e)
                     (list 'ALL (tag 'bound_expression 'b))))
      'ALL
  (if (rule e (prodn (tag 'expression 'e)
                     (list 'SOME (tag 'bound_expression 'b))))
      'SOME
    nil)))


(defn cdr_quantified_exp (e)
  (let ((qn (cdr_quantified_names e)))
    (if (equal qn nil)
        (bound_boolean_expression e)
      (mk_quantified_expression (quantifier e) qn (bound_id_type e)
                                (bound_boolean_expression e)))))


(defn constant_value_exp (u)

  (if (rule u (prodn (tag 'constant_declaration 'd)
                     (list 'CONST (tag 'IDENTIFIER 'cn)
                           'COLON (tag 'type_specification 'rt)
                           'COLON_EQUAL (tag 'constant_body 'b))))
      (constant_value_exp (subtree u 'constant_body))

  (if (rule u (prodn (tag 'constant_body 'b)
                     'PENDING))
      nil

  (if (rule u (prodn (tag 'constant_body 'b)
                     (tag 'expression 'p)))
      (subtree u 'expression)

    nil)))

  ( (lessp (tree_size u)) ))
```

```
(defn scalar_const_units (n u q)
  ;; n is an identifier, q a numberp.

  (if (rule u (prodn (tag 'identifier_list 'is)
                     (tag 'IDENTIFIER 'i)))
      (rcons nil
             (mk_scalar_const_unit n
                                   (gname (subtree u 'IDENTIFIER))
                                   q))

  (if (rule u (prodn (tag 'identifier_list 'is)
                     (list (tag 'identifier_list 'is2) 'COMMA
                           (tag 'IDENTIFIER 'i))))
      (rcons (scalar_const_units n (subtree u 'identifier_list) (sub1 q))
             (mk_scalar_const_unit n
                                   (gname (subtree u 'IDENTIFIER))
                                   q))

    nil))

  ( (lessp (tree_size u)) ))


(defn scalar_value_list (u)

  (if (rule u (prodn (tag 'type_declaration 'd)
                     (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
                           (tag 'type_definition 'd2))))
      (scalar_value_list (subtree u 'type_definition))

  (if (rule u (prodn (tag 'type_definition 'd)
                     (tag 'scalar_type 's)))
      (scalar_value_list (subtree u 'scalar_type))

  (if (rule u (prodn (tag 'scalar_type 's)
                     (list 'OPEN_PAREN (tag 'identifier_list 'is)
                           'CLOSE_PAREN)))
      (scalar_value_list (subtree u 'identifier_list))

  (if (rule u (prodn (tag 'identifier_list 'is)
                     (tag 'IDENTIFIER 'i)))
      (rcons nil (gname (subtree u 'IDENTIFIER)))

  (if (rule u (prodn (tag 'identifier_list 'is)
                     (list (tag 'identifier_list 'is2) 'COMMA
                           (tag 'IDENTIFIER 'i))))
      (rcons (scalar_value_list (subtree u 'identifier_list))
             (gname (subtree u 'IDENTIFIER)))

    nil)))))

  ( (lessp (tree_size u)) ))


(defn derived_units (n u)
  ;; n is an identifier.

  (if (rule u (prodn (tag 'type_definition 'd)
                     (tag 'scalar_type 's)))
      (derived_units n (subtree u 'scalar_type))

  (if (rule u (prodn (tag 'scalar_type 's)
                     (list 'OPEN_PAREN (tag 'identifier_list 'is)
                           'CLOSE_PAREN)))
      (scalar_const_units n (subtree u 'identifier_list)
                          (sub1 (length (scalar_value_list u))))

    nil))
```

```
  ( (lessp (tree_size u)) ))


(defn dparam_name (d)

  (if (rule d (prodn (tag 'similar_formal_data_parameters 'd)
                     (list (tag 'opt_access_specification 'a)
                           (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ft))))
      (dparam_name (subtree d 'identifier_list))

  (if (rule d (prodn (tag 'identifier_list 'is)
                     (tag 'IDENTIFIER 'i)))
      (dparam_name (subtree d 'IDENTIFIER))

  (if (identifierp d)
      (gname d)

    nil)))

  ( (lessp (tree_size d)) ))


(defn dparam_name_list (fs)
  (if (nlistp fs)
      nil
    (cons (dparam_name (car fs))
          (dparam_name_list (cdr fs)))))


(defn each_clausep (e)

  (if (rule e (prodn (tag 'opt_each_clause 'e)
                     (list 'EACH (tag 'IDENTIFIER 'i) 'COLON
                           (tag 'type_specification 'ts) 'COMMA)))
      T

  (if (rule e (prodn (tag 'opt_each_clause 'e)
                     'empty))
      F

    F)))


(defn expression_from_spec (e)

  (if (or (rule e (prodn (tag 'specification_expression 'se)
                         (tag 'validated_specification_expression 'se2)))
          (rule e (prodn (tag 'specification_expression 'se)
                         (list 'OPEN_PAREN
                               (tag 'validated_specification_expression 'se2)
                               'CLOSE_PAREN))))
      (expression_from_spec
         (subtree e 'validated_specification_expression))

  (if (rule e (prodn (tag 'validated_specification_expression 'se)
                     (list (tag 'non_validated_specification_expression 'se2)
                           'OTHERWISE (tag 'IDENTIFIER 'i))))
      (expression_from_spec
         (subtree e 'non_validated_specification_expression))

  (if (rule e (prodn (tag 'specification_expression 'se)
                     (tag 'non_validated_specification_expression 'se2)))
      (expression_from_spec
         (subtree e 'non_validated_specification_expression))

  (if (or (rule e (prodn (tag 'non_validated_specification_expression 'se)
                         (list 'OPEN_PAREN (tag 'proof_directive 'd)
                               (tag 'expression 'e) 'CLOSE_PAREN)))
```

```
          (rule e (prodn (tag 'non_validated_specification_expression 'se)
                         (list (tag 'proof_directive 'd)
                               (tag 'expression 'e))))
          (rule e (prodn (tag 'non_validated_specification_expression 'se)
                         (tag 'expression 'e))))
      (subtree e 'expression)

    nil))))

  ( (lessp (tree_size e)) ))


(defn object_namep (m)

  (if (rule m (prodn (tag 'expression 'e)
                     (tag 'modified_primary_value 'm)))
      (object_namep (subtree m 'modified_primary_value))

  (if (rule m (prodn (tag 'modified_primary_value 'm)
                     (tag 'primary_value 'p)))
      (object_namep (subtree m 'primary_value))

  (if (rule m (prodn (tag 'primary_value 'p)
                     (tag 'IDENTIFIER 'i)))
      T

    F)))

  ( (lessp (tree_size m)) ))


(defn fn_call_formp (m)

  (if (rule m (prodn (tag 'expression 'e)
                     (tag 'modified_primary_value 'm)))
      (fn_call_formp (subtree m 'modified_primary_value))

  (if (rule m (prodn (tag 'modified_primary_value 'm)
                     (tag 'primary_value 'p)))
      (object_namep m)

  (if (rule m (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'value_modifiers 'vm))))
      (and (object_namep (subtree m 'modified_primary_value))
           (arg_listp (subtree m 'value_modifiers)))

  (if (rule m (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'actual_condition_parameters 'cp))))
      (fn_call_formp (subtree m 'modified_primary_value))

    F))))

  ( (lessp (tree_size m)) ))


(defn foreign_name (u)

  (if (rule u (prodn (tag 'name_declaration 'd)
                     (list 'NAME (tag 'local_aliases 'a)
                           'FROM (tag 'IDENTIFIER 'fs))))
      ; The IDENTIFIER is a foreign_scope_name.
      (foreign_name (subtree u 'local_aliases))

  (if (rule u (prodn (tag 'local_aliases 'a)
                     (tag 'local_renaming 'r)))
      (foreign_name (subtree u 'local_renaming))
```

```
  (if (rule u (prodn (tag 'local_aliases 'a)
                     (list (tag 'local_aliases 'a2) 'COMMA
                           (tag 'local_renaming 'r))))
      (foreign_name (subtree u 'local_renaming))

  (if (rule u (prodn (tag 'local_renaming 'r)
                     (tag 'IDENTIFIER 'fn)))
      ; The IDENTIFIER is a foreign_unit_name.
      (foreign_name (subtree u 'IDENTIFIER))

  (if (rule u (prodn (tag 'local_renaming 'r)
                     (list (tag 'IDENTIFIER 'ln) 'EQUAL
                           (tag 'IDENTIFIER 'fn))))
      ; The first IDENTIFIER is a local_name.
      ; The second IDENTIFIER is a foreign_unit_name.
      (foreign_name (subtree_i u 'IDENTIFIER 2))

  (if (identifierp u)
      (gname u)

    nil))))))

  ( (lessp (tree_size u)) ))


(defn foreign_scope_name (u)

  (if (rule u (prodn (tag 'name_declaration 'd)
                     (list 'NAME (tag 'local_aliases 'a)
                           'FROM (tag 'IDENTIFIER 'fs))))
      ; The IDENTIFIER is a foreign_scope_name.
      (foreign_scope_name (subtree u 'IDENTIFIER))

  (if (identifierp u)
      (gname u)

    nil))

  ( (lessp (tree_size u)) ))


(defn full_dargs (a p ft)
  ; a is 'CONST or 'VAR. ft is type_specification

  (if (rule p (prodn (tag 'identifier_list 'is)
                     (tag 'IDENTIFIER 'i)))
      (rcons nil
             (mk_single_formal_data_parameter a (subtree p 'IDENTIFIER) ft))

  (if (rule p (prodn (tag 'identifier_list 'is)
                     (list (tag 'identifier_list 'is2) 'COMMA
                           (tag 'IDENTIFIER 'i))))
      (rcons (full_dargs a (subtree p 'identifier_list) ft)
             (mk_single_formal_data_parameter a (subtree p 'IDENTIFIER) ft))

    nil))

  ( (lessp (tree_size p)) ))


(defn formal_dargs (d)

  (if (rule d (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (formal_dargs (subtree d 'external_data_objects))
```

```
  (if (rule d (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
                           'COLON (tag 'type_specification 'rt)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (formal_dargs (subtree d 'opt_external_data_objects))

  (if (rule d (prodn (tag 'opt_external_data_objects 'd)
                     'empty))
      nil

  (if (rule d (prodn (tag 'opt_external_data_objects 'd)
                     (tag 'external_data_objects 'd2)))
      (formal_dargs (subtree d 'external_data_objects))

  (if (rule d (prodn (tag 'external_data_objects 'd)
                     (list 'OPEN_PAREN (tag 'external_data_objects_list 'd2)
                           'CLOSE_PAREN)))
      (formal_dargs (subtree d 'external_data_objects_list))

  (if (rule d (prodn (tag 'external_data_objects_list 'd)
                     (tag 'similar_formal_data_parameters 'd2)))
      (formal_dargs (subtree d 'similar_formal_data_parameters))

  (if (rule d (prodn (tag 'external_data_objects_list 'd)
                     (list (tag 'external_data_objects_list 'd2) 'SEMI_COLON
                           (tag 'similar_formal_data_parameters 'd3))))
      (append (formal_dargs (subtree d 'external_data_objects_list))
              (formal_dargs (subtree d 'similar_formal_data_parameters)))

  (if (rule d (prodn (tag 'similar_formal_data_parameters 'd)
                     (list (tag 'opt_access_specification 'a)
                           (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ft))))
      (full_dargs (access (subtree d 'opt_access_specification))
                  (subtree d 'identifier_list)
                  (subtree d 'type_specification))

  (if (rule d (prodn (tag 'constant_declaration 'd)
                     (list 'CONST (tag 'IDENTIFIER 'cn)
                           'COLON (tag 'type_specification 'rt)
                           'COLON_EQUAL (tag 'constant_body 'b))))
      nil

    nil)))))))))

  ( (lessp (tree_size d)) ))


(defn formal_type (d)

  (if (rule d (prodn (tag 'similar_formal_data_parameters 'd)
                     (list (tag 'opt_access_specification 'a)
                           (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ft))))
      (subtree d 'type_specification)

    nil))


(defn ibase (b)

  (if (rule b (prodn (tag 'base 'b) 'BINARY))
      2

  (if (rule b (prodn (tag 'base 'b) 'OCTAL))
      8
```

```
  (if (rule b (prodn (tag 'base 'b) 'DECIMAL))
      10

  (if (rule b (prodn (tag 'base 'b) 'HEX))
      16

    nil)))))


(defn if_else_exp (e)

  (if (rule e (prodn (tag 'if_expression 'i)
                     (list 'IF (tag 'expression 'b) 'THEN
                           (tag 'expression 'p)
                           (tag 'if_expression_else_part 'e))))
      (if_else_exp (subtree e 'if_expression_else_part))

  (if (rule e (prodn (tag 'if_expression_else_part 'e)
                     (list 'ELSE (tag 'expression 'p) 'FI)))
      (subtree e 'expression)

  (if (rule e (prodn (tag 'if_expression_else_part 'e)
                     (list 'ELIF (tag 'expression 'b) 'THEN
                           (tag 'expression 'p)
                           (tag 'if_expression_else_part 'e2))))
      (mk_tree 'if_expression
               (cons (mk_reserved_word 'IF) (cdr (subtrees e))))

    nil)))
  ( (lessp (tree_size e)) ))


(defn kind (u)

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'type_declaration 'd2)))
      (kind (subtree u 'type_declaration))

  (if (rule u (prodn (tag 'type_declaration 'd)
                     (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
                           (tag 'type_definition 'd2))))
      'type

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'procedure_declaration 'd2)))
      (kind (subtree u 'procedure_declaration))

  (if (rule u (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      'procedure

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'function_declaration 'd2)))
      (kind (subtree u 'function_declaration))

  (if (rule u (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
                           'COLON (tag 'type_specification 'rt)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      'function

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'constant_declaration 'd2)))
      (kind (subtree u 'constant_declaration))
```

```
  (if (rule u (prodn (tag 'constant_declaration 'd)
                     (list 'CONST (tag 'IDENTIFIER 'cn)
                           'COLON (tag 'type_specification 'rt)
                           'COLON_EQUAL (tag 'constant_body 'b))))
      'constant

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'lemma_declaration 'd2)))
      (kind (subtree u 'lemma_declaration))

  (if (rule u (prodn (tag 'lemma_declaration 'd)
                     (list 'LEMMA (tag 'IDENTIFIER 'ln)
                           (tag 'opt_external_data_objects 'a)
                           'EQUAL
                           (tag 'non_validated_specification_expression 'b))))
      'lemma

  (if (rule u (prodn (tag 'name_declaration 'd)
                     (list 'NAME (tag 'local_aliases 'a)
                           'FROM (tag 'IDENTIFIER 'fs))))
      ; The IDENTIFIER is a foreign_scope_name.
      'name

  (if (errorp u)
      'error

    nil))))))))))))

  ( (lessp (tree_size u)) ))


(defn unit_name (u)

  (if (rule u (prodn (tag 'type_declaration 'd)
                     (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
                           (tag 'type_definition 'd2))))
      (unit_name (subtree u 'IDENTIFIER))

  (if (rule u (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (unit_name (subtree u 'IDENTIFIER))

  (if (rule u (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
                           'COLON (tag 'type_specification 'rt)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (unit_name (subtree u 'IDENTIFIER))

  (if (rule u (prodn (tag 'constant_declaration 'd)
                     (list 'CONST (tag 'IDENTIFIER 'cn)
                           'COLON (tag 'type_specification 'rt)
                           'COLON_EQUAL (tag 'constant_body 'b))))
      (unit_name (subtree u 'IDENTIFIER))

  (if (rule u (prodn (tag 'lemma_declaration 'd)
                     (list 'LEMMA (tag 'IDENTIFIER 'ln)
                           (tag 'opt_external_data_objects 'a)
                           'EQUAL
                           (tag 'non_validated_specification_expression 'b))))
      (unit_name (subtree u 'IDENTIFIER))

  (if (identifierp u)
      (gname u)
```

```
    nil))))))

  ( (lessp (tree_size u)) ))


(defn local_name (u)

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'type_declaration 'd2)))
      (unit_name (subtree u 'type_declaration))

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'procedure_declaration 'd2)))
      (unit_name (subtree u 'procedure_declaration))

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'function_declaration 'd2)))
      (unit_name (subtree u 'function_declaration))

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'constant_declaration 'd2)))
      (unit_name (subtree u 'constant_declaration))

  (if (rule u (prodn (tag 'unit_declaration 'd)
                     (tag 'lemma_declaration 'd2)))
      (unit_name (subtree u 'lemma_declaration))

  (if (rule u (prodn (tag 'name_declaration 'd)
                     (list 'NAME (tag 'local_aliases 'a)
                           'FROM (tag 'IDENTIFIER 'fs))))
      ; The IDENTIFIER is a foreign_scope_name.
      (local_name (subtree u 'local_aliases))

  (if (rule u (prodn (tag 'local_aliases 'a)
                     (tag 'local_renaming 'r)))
      (local_name (subtree u 'local_renaming))

  (if (rule u (prodn (tag 'local_aliases 'a)
                     (list (tag 'local_aliases 'a2) 'COMMA
                           (tag 'local_renaming 'r))))
      (local_name (subtree u 'local_renaming))

  (if (rule u (prodn (tag 'local_renaming 'r)
                     (tag 'IDENTIFIER 'fn)))
      ; The IDENTIFIER is a foreign_unit_name.
      (local_name (subtree u 'IDENTIFIER))

  (if (rule u (prodn (tag 'local_renaming 'r)
                     (list (tag 'IDENTIFIER 'ln) 'EQUAL
                           (tag 'IDENTIFIER 'fn))))
      ; The first IDENTIFIER is a local_name.
      ; The second IDENTIFIER is a foreign_unit_name.
      (local_name (subtree_i u 'IDENTIFIER 1))

  (if (identifierp u)
      (gname u)

    (unit_name u))))))))))))

  ( (lessp (tree_size u)) ))


(defn named_unit (u s)
  (mk_named_unit u s))


(defn named_unit_list (u fs)
  ; fs is an IDENTIFIER
```

```
  (if (rule u (prodn (tag 'local_aliases 'a)
                     (tag 'local_renaming 'r)))
      (rcons nil (named_unit (subtree u 'local_renaming) fs))

  (if (rule u (prodn (tag 'local_aliases 'a)
                     (list (tag 'local_aliases 'a2) 'COMMA
                           (tag 'local_renaming 'r))))
      (rcons (named_unit_list (subtree u 'local_aliases) fs)
             (named_unit (subtree u 'local_renaming) fs))

    nil))

  ( (lessp (tree_size u)) ))


(defn object_name (e)

  (if (rule e (prodn (tag 'expression 'e)
                     (tag 'modified_primary_value 'm)))
      (object_name (subtree e 'modified_primary_value))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
                     (tag 'primary_value 'p)))
      (object_name (subtree e 'primary_value))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'value_modifiers 'vm))))
      (object_name (subtree e 'modified_primary_value))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'actual_condition_parameters 'cp))))
      (object_name (subtree e 'modified_primary_value))

  (if (rule e (prodn (tag 'primary_value 'p)
                     (tag 'IDENTIFIER 'i)))
      (object_name (subtree e 'IDENTIFIER))

  (if (identifierp e)
      (gname e)

    nil))))))

  ( (lessp (tree_size e)) ))


(defn pending_type_defnp (d)

  (if (rule d (prodn (tag 'type_declaration 'd)
                     (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
                           (tag 'type_definition 'd2))))
      (pending_type_defnp (subtree d 'type_definition))

  (if (rule d (prodn (tag 'type_definition 'd)
                     'PENDING))
      T

    F))

  ( (lessp (tree_size d)) ))


(defn postc (u c)
  (let ((e (case_exit_list u c)))
    (if (equal (length e) 0)
        (mk_true_expression)
    (if (equal (length e) 1)
        (expression_from_spec (car e))
```

```
      (many_post_conditions_error u c)))))


(defn prec (u)

  (if (rule u (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (prec (subtree u 'procedure_body))

  (if (rule u (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
                           'COLON (tag 'type_specification 'rt)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (prec (subtree u 'procedure_body))

  (if (rule u (prodn (tag 'constant_declaration 'd)
                     (list 'CONST (tag 'IDENTIFIER 'cn)
                           'COLON (tag 'type_specification 'rt)
                           'COLON_EQUAL (tag 'constant_body 'b))))
      (mk_true_expression)

  (if (rule u (prodn (tag 'procedure_body 'b)
                     'PENDING))
      (mk_true_expression)

  (if (rule u (prodn (tag 'procedure_body 'b)
                     (list 'BEGIN
                           (tag 'external_operational_specification 'es)
                           (tag 'opt_internal_environment 'iv)
                           (tag 'opt_keep_specification 'k)
                           (tag 'opt_internal_statements 'st)
                           'END)))
      (prec (subtree u 'external_operational_specification))

  (if (rule u (prodn (tag 'external_operational_specification 's)
                     (list (tag 'opt_entry_specification 'e)
                           (tag 'opt_exit_specification 'x))))
      (prec (subtree u 'opt_entry_specification))

  (if (rule u (prodn (tag 'opt_entry_specification 'e)
                     'empty))
      (mk_true_expression)

  (if (rule u (prodn (tag 'opt_entry_specification 'e)
                     (list 'ENTRY
                           (tag 'non_validated_specification_expression
                                'se)
                           'SEMI_COLON)))
      (prec (subtree u 'non_validated_specification_expression))

  (if (or (rule u (prodn (tag 'non_validated_specification_expression 'se)
                         (list 'OPEN_PAREN (tag 'proof_directive 'd)
                               (tag 'expression 'e) 'CLOSE_PAREN)))
          (rule u (prodn (tag 'non_validated_specification_expression 'se)
                         (list (tag 'proof_directive 'd)
                               (tag 'expression 'e))))
          (rule u (prodn (tag 'non_validated_specification_expression 'se)
                         (tag 'expression 'e))))
      (subtree u 'expression)

    nil)))))))))

  ( (lessp (tree_size u)) ))
```

```
(defn record_field_names (d)

  (if (rule d (prodn (tag 'record_type 'r)
                     (list 'RECORD 'OPEN_PAREN (tag 'fields 'f)
                           'CLOSE_PAREN)))
      (record_field_names (subtree d 'fields))

  (if (rule d (prodn (tag 'fields 'f)
                     (list (tag 'fields 'f2) 'SEMI_COLON
                           (tag 'similar_fields 's))))
      (append (record_field_names (subtree d 'fields))
              (record_field_names (subtree d 'similar_fields)))

  (if (rule d (prodn (tag 'fields 'f)
                     (tag 'similar_fields 's)))
      (record_field_names (subtree d 'similar_fields))

  (if (rule d (prodn (tag 'similar_fields 's)
                     (list (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ft))))
      (record_field_names (subtree d 'identifier_list))

  (if (rule d (prodn (tag 'identifier_list 'is)
                     (list (tag 'identifier_list 'is2) 'COMMA
                           (tag 'IDENTIFIER 'i))))
      (rcons (record_field_names (subtree d 'identifier_list))
             (record_field_names (subtree d 'IDENTIFIER)))

  (if (rule d (prodn (tag 'identifier_list 'is)
                     (tag 'IDENTIFIER 'i)))
      (rcons nil (record_field_names (subtree d 'IDENTIFIER)))

  (if (identifierp d)
      (gname d)

    nil)))))))

  ( (lessp (tree_size d)) ))


(defn result_type (d)

  (if (rule d (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
                           'COLON (tag 'type_specification 'rt)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (subtree d 'type_specification)

  (if (rule d (prodn (tag 'constant_declaration 'd)
                     (list 'CONST (tag 'IDENTIFIER 'cn)
                           'COLON (tag 'type_specification 'rt)
                           'COLON_EQUAL (tag 'constant_body 'b))))
      (subtree d 'type_specification)

    nil)))


(defn scalar_type_defnp (d)

  (if (rule d (prodn (tag 'type_declaration 'd)
                     (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
                           (tag 'type_definition 'd2))))
      (scalar_type_defnp (subtree d 'type_definition))

  (if (rule d (prodn (tag 'type_definition 'd)
                     (tag 'scalar_type 's)))
      T
```

```
    F))

  ( (lessp (tree_size d)) ))


(defn scope_list (u)

  (if (rule u (prodn (tag 'program_description 'pd)
                     (list (tag 'scope_declaration_list 'ss)
                           'opt_semi_colon)))
      (scope_list (subtree u 'scope_declaration_list))

  (if (rule u (prodn (tag 'scope_declaration_list 'ss)
                     (tag 'scope_declaration 'sd)))
      (rcons nil (subtree u 'scope_declaration))

  (if (rule u (prodn (tag 'scope_declaration_list 'ss)
                     (list (tag 'scope_declaration_list 'ss2) 'SEMI_COLON
                           (tag 'scope_declaration 'sd))))
      (rcons (scope_list (subtree u 'scope_declaration_list))
             (subtree u 'scope_declaration))

    nil)))

  ( (lessp (tree_size u)) ))


(defn scope_name (u)

  (if (rule u (prodn (tag 'scope_declaration 'sd)
                     (list 'SCOPE (tag 'IDENTIFIER 'sn) 'EQUAL
                           'BEGIN (tag 'unit_or_name_declaration_list 'ul)
                           'opt_semi_colon
                           'END)))
      (scope_name (subtree u 'IDENTIFIER))

  (if (identifierp u)
      (gname u)

    nil))

  ( (lessp (tree_size u)) ))


(defn unit_list (u)

  (if (rule u (prodn (tag 'scope_declaration 'sd)
                     (list 'SCOPE (tag 'IDENTIFIER 'sn) 'EQUAL
                           'BEGIN (tag 'unit_or_name_declaration_list 'ul)
                           'opt_semi_colon
                           'END)))
      (unit_list (subtree u 'unit_or_name_declaration_list))

  (if (rule u (prodn (tag 'unit_or_name_declaration_list 'us)
                     (tag 'unit_or_name_declaration 'd)))
      (unit_list (subtree u 'unit_or_name_declaration))

  (if (rule u (prodn (tag 'unit_or_name_declaration_list 'us)
                     (list (tag 'unit_or_name_declaration_list 'us2)
                           'SEMI_COLON
                           (tag 'unit_or_name_declaration 'd))))
      (append (unit_list (subtree u 'unit_or_name_declaration_list))
              (unit_list (subtree u 'unit_or_name_declaration)))

  (if (rule u (prodn (tag 'unit_or_name_declaration 'd)
                     (tag 'unit_declaration 'd2)))
      (unit_list (subtree u 'unit_declaration))

  (if (rule u (prodn (tag 'unit_declaration 'd)
```

```
                   (tag 'type_declaration 'd2)))
    (unit_list (subtree u 'type_declaration))

(if (rule u (prodn (tag 'type_declaration 'd)
                   (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
                         (tag 'type_definition 'd2))))
    (cons u (derived_units (subtree u 'IDENTIFIER)
                           (subtree u 'type_definition)))

(if (rule u (prodn (tag 'unit_declaration 'd)
                   (tag 'procedure_declaration 'd2)))
    (unit_list (subtree u 'procedure_declaration))

(if (rule u (prodn (tag 'procedure_declaration 'd)
                   (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                         (tag 'external_data_objects 'a)
                         (tag 'opt_external_conditions 'c)
                         'EQUAL (tag 'procedure_body 'b))))
    (cons u nil)

(if (rule u (prodn (tag 'unit_declaration 'd)
                   (tag 'function_declaration 'd2)))
    (unit_list (subtree u 'function_declaration))

(if (rule u (prodn (tag 'function_declaration 'd)
                   (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                         (tag 'opt_external_data_objects 'a)
                         'COLON (tag 'type_specification 'rt)
                         (tag 'opt_external_conditions 'c)
                         'EQUAL (tag 'procedure_body 'b))))
    (cons u nil)

(if (rule u (prodn (tag 'unit_declaration 'd)
                   (tag 'constant_declaration 'd2)))
    (unit_list (subtree u 'constant_declaration))

(if (rule u (prodn (tag 'constant_declaration 'd)
                   (list 'CONST (tag 'IDENTIFIER 'cn)
                         'COLON (tag 'type_specification 'rt)
                         'COLON_EQUAL (tag 'constant_body 'b))))
    (cons u nil)

(if (rule u (prodn (tag 'unit_declaration 'd)
                   (tag 'lemma_declaration 'd2)))
    (unit_list (subtree u 'lemma_declaration))

(if (rule u (prodn (tag 'lemma_declaration 'd)
                   (list 'LEMMA (tag 'IDENTIFIER 'ln)
                         (tag 'opt_external_data_objects 'a)
                         'EQUAL
                         (tag 'non_validated_specification_expression 'b))))
    (cons u nil)

(if (rule u (prodn (tag 'unit_or_name_declaration 'd)
                   (tag 'name_declaration 'd2)))
    (unit_list (subtree u 'name_declaration))

(if (rule u (prodn (tag 'name_declaration 'd)
                   (list 'NAME (tag 'local_aliases 'a)
                         'FROM (tag 'IDENTIFIER 'fs))))
    ; The IDENTIFIER is a foreign_scope_name.
    (named_unit_list (subtree u 'local_aliases)
                     (subtree u 'IDENTIFIER))

  nil)))))))))))))))

( (lessp (tree_size u)) ))
```

```
; **********************
;  Reserved Identifiers
; **********************

; (defn reserved_words () ...)
;   defined above under "Parse Tree Leaves"

(defn standard_ids ()
  '(activationid boolean character integer rational

    true false

    allfrom allto content domain empty first full infrom
    infrommerge initial last lower max messages min nonfirst nonlast
    null ord outto outtomerge pred range scale size succ
    timedallfrom timedallto timedinfrom timedinfrommerge timedmerge
    timedorder timedoutto timedouttomerge upper

    routineerror spaceerror))


(defn reserved_idp (n)
  (let ((i (if (litatom n)
               (pack (uc_list (unpack n)))
             (pack (uc_list n)))))
    (or (member i (reserved_words))
        (member i (standard_ids)))))

(defn some_reserved_idp (s)
  (if (nlistp s)
      F
    (or (reserved_idp (car s))
        (some_reserved_idp (cdr s)))))



;; *****************
;;  Unit references
;; *****************

(defn local_unit_names (ul)
  (if (nlistp ul)
      nil
    (cons (local_name (car ul))
          (local_unit_names (cdr ul)))))

(defn local_names (sd)
  (cons (scope_name sd)
        (local_unit_names (unit_list sd))))

(defn all_scopes (n s)
  (if (nlistp s)
      nil
    (if (equal (scope_name (car s)) n)
        (cons (car s) (all_scopes n (cdr s)))
      (all_scopes n (cdr s)))))

(defn all_units (n u)
  (if (nlistp u)
      nil
    (if (equal (local_name (car u)) n)
        (cons (car u) (all_units n (cdr u)))
      (all_units n (cdr u)))))

(disable reserved_idp)
(disable unit_reserved_error)
(disable scope_reserved_error)
(disable scope_id_error)
(disable alias_id_error)
```

```
(disable no_scope_error)
(disable no_unit_error)
(disable many_unit_error)
(disable many_scope_error)

(defn mref (i sn x m)
  (if (reserved_idp i)
      (cons sn (unit_reserved_error i))
  (if (reserved_idp sn)
      (cons sn (scope_reserved_error sn))
  (if (equal i sn)
      (cons sn (scope_id_error sn))
  (if (equal (fix m) 0)
      (cons sn (alias_id_error i sn))
    (let ((slist (all_scopes sn (scope_list x))))
      (if (equal (length slist) 0)
          (cons sn (no_scope_error sn))
      (if (equal (length slist) 1)
          (let ((ulist (all_units i (unit_list (car slist)))))
            (if (equal (length ulist) 0)
                (cons sn (no_unit_error i sn))
            (if (equal (length ulist) 1)
                (let ((u (car ulist)))
                  (if (equal (kind u) 'name)
                      (mref (foreign_name u) (foreign_scope_name u)
                            x (sub1 m))
                    (cons sn u)))
              (cons sn (many_unit_error i sn)))))
        (cons sn (many_scope_error sn))))))))))

(enable reserved_idp)
(enable unit_reserved_error)
(enable scope_reserved_error)
(enable scope_id_error)
(enable alias_id_error)
(enable no_scope_error)
(enable no_unit_error)
(enable many_unit_error)
(enable many_scope_error)


(defn ref (i sn x)
  (mref i sn x 2))

(defn ref_unit (x)
  (cdr x))

(defn ref_scope (x)
  (car x))



; ****************
;  Marked Objects
; ****************

(add-shell marked nil markedp
           ((mark (none-of) false)
            (object (none-of) zero)))

(defn unmark (o)
  (if (markedp o)
      (object o)
    o))



; ***************************
;  Pre-Computable Expressions
```

```
; ****************************

(dcl precomputable_F (e c x))
; constraint on precomputable_F:
;   (or (indeterminate (precomputable_F e c x))
;       (equal (precomputable_F e c x)
;              (gF e c (empty_map) n x)))
; for some n.



; ****************************
;  Type Descriptor Primitives
; ****************************

; A type descriptor is a tree with different forms for integers/rationals,
; scalar types, arrays, records, mappings, sequences, and sets.  There are
; also special forms for pending types and errors.

; =======================
;  Primitive Constructors
; =======================

; -------------
;  Simple Types
; -------------

(defn mk_integer_desc (tmin tmax udv)
  ; Constructs an integer type descriptor td, if
  ;     tmin is the untyped value for lower(td) or nil if unbounded or
  ;          an errorp
  ;     tmax is the untyped value for upper(td) or nil if unbounded or
  ;          an errorp
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'integer
           (list (mk_tree 'tmin tmin)
                 (mk_tree 'tmax tmax)
                 (mk_tree 'udv udv))))

(defn mk_rational_desc (tmin tmax udv)
  ; Constructs a rational type descriptor td, if
  ;     tmin is the untyped value for lower(td) or nil if unbounded or
  ;          an errorp
  ;     tmax is the untyped value for upper(td) or nil if unbounded or
  ;          an errorp
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'rational
           (list (mk_tree 'tmin tmin)
                 (mk_tree 'tmax tmax)
                 (mk_tree 'udv udv))))

(defn mk_scalar_desc (tid sid crd tmin tmax udv)
  ; Constructs a scalar type descriptor td, if
  ;     tid is a type name (litatom),
  ;     sid is the name (litatom) of the scope where tid is declared
  ;         or nil for standard types,
  ;     crd is the cardinality of the value set of the base type of tid
  ;     tmin is the untyped value for lower(td) or an errorp
  ;     tmax is the untyped value for upper(td) or an errorp
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'scalar
           (list (mk_tree 'tid tid)
                 (mk_tree 'sid sid)
                 (mk_tree 'crd crd)
                 (mk_tree 'tmin tmin)
                 (mk_tree 'tmax tmax)
                 (mk_tree 'udv udv))))
```

```
; -------------
;  Array Types
; -------------

(defn mk_array_desc (id cd udv)
  ; Constructs an array type descriptor td, if
  ;     id is the type descriptor for the array index type
  ;     cd is the type descriptor for the array component type
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'array
           (list (mk_tree 'selector_td id)
                 (mk_tree 'component_td cd)
                 (mk_tree 'udv udv))))


; --------------
;  Record Types
; --------------

(defn mk_record_desc (fds udv)
  ; Constructs a record type descriptor td, if
  ;     fds is a name-value mapping from field names to their type descriptors
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'record
           (list (mk_tree 'field_tds fds)
                 (mk_tree 'udv udv))))


; ---------------
;  Mapping Types
; ---------------

(defn mk_mapping_desc (sl sd cd udv)
  ; Constructs a mapping type descriptor td, if
  ;     sl is the size limit restriction of the mapping type, nil or
  ;        numberp or errorp
  ;     sd is the type descriptor for the mapping selector type
  ;     cd is the type descriptor for the mapping component type
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'mapping
           (list (mk_tree 'max_size sl)
                 (mk_tree 'selector_td sd)
                 (mk_tree 'component_td cd)
                 (mk_tree 'udv udv))))


; ----------------
;  Sequence Types
; ----------------

(defn mk_sequence_desc (sl cd udv)
  ; Constructs a sequence type descriptor td, if
  ;     sl is the size limit restriction of the sequence type, nil or
  ;        numberp or errorp
  ;     cd is the type descriptor for the sequence component type
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'sequence
           (list (mk_tree 'max_size sl)
                 (mk_tree 'component_td cd)
                 (mk_tree 'udv udv))))


; -----------
;  Set Types
; -----------

(defn mk_set_desc (sl cd udv)
  ; Constructs a set type descriptor td, if
  ;     sl is the size limit restriction of the set type, nil or
```

```
  ;        numberp or errorp
  ;     cd is the type descriptor for the set component type
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'set
           (list (mk_tree 'max_size sl)
                 (mk_tree 'component_td cd)
                 (mk_tree 'udv udv))))


; --------------
;  Pending Types
; --------------

(defn mk_pending_desc (tid sid udv)
  ; Constructs a type descriptor td for a pending type, if
  ;     tid is the type name (litatom)
  ;     sid is the name (litatom) of the scope where tid is declared
  ;     udv is the untyped value for initial(td) or an errorp
  (mk_tree 'pending
           (list (mk_tree 'tid tid)
                 (mk_tree 'sid sid)
                 (mk_tree 'udv udv))))


; ------------
;  Type Errors
; ------------

; The type descriptor constructor for type errors is
;
;     mk_error (<error message>)
;
; It and functions for specific type errors are defined in the section on
; errors, above.



; =====================
;  Primitive Extractors
; =====================

(defn component_td (td)
  (subtree_body td 'component_td))

(defn crd (td)
  (subtree_body td 'crd))

(defn field_tds (td)
  (subtree_body td 'field_tds))

(defn field_td (fn td)
  (mapped_value (field_tds td) fn))

(defn max_size (td)
  (subtree_body td 'max_size))

(defn selector_td (td)
  (subtree_body td 'selector_td))

(defn sid (td)
  (subtree_body td 'sid))

(defn tid (td)
  (if (member (root td) '(integer rational))
      (root td)
    (subtree_body td 'tid)))

(defn tmax (td)
  (subtree_body td 'tmax))
```

```
(defn tmin (td)
  (subtree_body td 'tmin))

(defn type_error_msg (td)
  (if (errorp td)
      (error_msg td)
    nil))

(defn udv (td)
  (if (errorp td)
      td
    (subtree_body td 'udv)))


; ======================
;  Primitive Recognizers
; ======================

(defn integer_descp (td)
  (equal td (mk_integer_desc (tmin td) (tmax td) (udv td))))

(defn rational_descp (td)
  (equal td (mk_rational_desc (tmin td) (tmax td) (udv td))))

(defn scalar_descp (td)
  (equal td (mk_scalar_desc (tid td) (sid td) (crd td)
                            (tmin td) (tmax td) (udv td))))

(defn simple_descp (td)
  (or (integer_descp td)
      (rational_descp td)
      (scalar_descp td)))

(defn array_descp (td)
  (equal td (mk_array_desc (selector_td td) (component_td td) (udv td))))

(defn record_descp (td)
  (equal td (mk_record_desc (field_tds td) (udv td))))

(defn mapping_descp (td)
  (equal td (mk_mapping_desc (max_size td) (selector_td td)
                             (component_td td) (udv td))))

(defn sequence_descp (td)
  (equal td (mk_sequence_desc (max_size td) (component_td td) (udv td))))

(defn set_descp (td)
  (equal td (mk_set_desc (max_size td) (component_td td) (udv td))))

(defn pending_descp (td)
  (equal td (mk_pending_desc (tid td) (sid td) (udv td))))

(defn error_descp (td)
  (errorp td))

(defn type_descp (td)
  (or (simple_descp td)
      (array_descp td)
      (record_descp td)
      (mapping_descp td)
      (sequence_descp td)
      (set_descp td)
      (pending_descp td)
      (error_descp td)))


; =========================
;  Setting Descriptor Parts
; =========================
```

```
(defn set_tmax (td v)
  (subst_tree (mk_tree ’tmax v)
              (subtree td ’tmax)
              td))

(defn set_tmin (td v)
  (subst_tree (mk_tree ’tmin v)
              (subtree td ’tmin)
              td))

(defn set_udv (td v)
  (subst_tree (mk_tree ’udv v)
              (subtree td ’udv)
              td))


; ***************************************
;  Descriptors for Standard Simple Types
; ***************************************

(defn boolean_desc ()
  (mk_scalar_desc ’boolean nil 2 0 1 0))

(defn character_desc ()
  (mk_scalar_desc ’character nil 128 0 127 0))

(defn integer_desc ()
  (mk_integer_desc nil nil 0))

(defn rational_desc ()
  (mk_rational_desc nil nil (rational 0 1)))



; ******************
;  Type Recognizers
; ******************

(defn integer_typep (td)
  (integer_descp td))

(defn rational_typep (td)
  (rational_descp td))

(defn scalar_typep (td)
  (scalar_descp td))

(defn boolean_typep (td)
  (and (scalar_typep td)
       (equal (tid td) ’boolean)))

(defn character_typep (td)
  (and (scalar_typep td)
       (equal (tid td) ’character)))

(defn simple_typep (td)
  (or (integer_typep td)
      (rational_typep td)
      (scalar_typep td)))

(defn bounded_typep (td)
  (if (simple_typep td)
      (and (not (equal (tmin td) nil))
           (not (equal (tmax td) nil)))
    F))

(defn equality_typep (td)
  ; all types are equality types so far
  (type_descp td))
```

```
(defn non_rational_simple_typep (td)
  (and (simple_typep td)
       (not (rational_typep td))))

(defn index_typep (td)
  (non_rational_simple_typep td))

(defn bounded_index_typep (td)
  (and (index_typep td)
       (bounded_typep td)))



; ***************************
;  Functions on Gypsy Values
; ***************************

; Gypsy values are represented as integers, rationals, selector-component
; maps, and lists.   A selector-component map is a list of pairs.  The car of
; each pair is a selector (index, field name).  The cdr is the Gypsy value of
; that component.
;
; Values of boolean, character, integer, and user-defined scalar types are all
; represented as integers.
;
; Values of rational types are represented as rationals.
;
; Values of arrays, records, and mappings are represented as
; selector-component maps from indexes, field names, and selectors,
; respectively, to components.
;
; Values of sequences and sets are represented as lists of components.


(defn null_map ()
  (empty_map))

(defn null_seq ()
  nil)

(defn null_set ()
  nil)

(defn field_names (v)
  (keys v))

(defn vselectors (v)
  ; v is a selector-component map
  (keys v))

(defn vcomponents (v)
  ; v is a selector-component map
  (key_values v))

(defn vdomain (v)
  (vselectors v))

(defn vindexes (v)
  (vselectors v))

(defn vrange (v)
  (vcomponents v))

(defn fselect (m k)
  ; k is a record field name
  ; m is a key-value map with field name keys
  (if (in_map m k)
      (mapped_value m k)
    (no_such_component_error m k)))
```

```
; ================================
;  Lemmas for the Vequal Do-Mutual
; ================================

(prove-lemma list_tree_size_not_zero (rewrite)
  (implies (listp x)
           (not (equal (tree_size x) 0))))

(prove-lemma lessp_list_subtree_than_subtrees (rewrite)
  (implies (listp x)
           (lessp (tree_size (list_subtree x n i))
                  (tree_size x))))

(prove-lemma lessp_subtree_than_subtrees (rewrite)
  (implies (listp (subtrees x))
           (lessp (tree_size (subtree x n))
                  (tree_size (subtrees x))))
  ( (enable subtree) ))

(prove-lemma lessp_subtree_body_than_subtrees (rewrite)
  (implies (listp (subtrees x))
           (lessp (tree_size (subtree_body x n))
                  (tree_size (subtrees x))))
  ( (enable subtree_body)
    (use (lessp_subtree_than_subtrees)) ))

(prove-lemma lessp_mapping_domain_tree_size_0 (rewrite)
  (lessp (tree_size (mk_set_desc nil sd nil))
         (tree_size (subtrees (mk_mapping_desc s sd cd dv))))
  ( (enable subtree_body subtree list_subtree) ))

(prove-lemma lessp_mapping_domain_tree_size (rewrite)
  (implies (mapping_descp x)
           (lessp (tree_size (mk_set_desc nil (subtree_body x 'selector_td)
                                          nil))
                  (tree_size (subtrees x))))
  ( (disable mk_set_desc mk_mapping_desc)
    (use (lessp_mapping_domain_tree_size_0 (s (max_size x))
                                           (sd (selector_td x))
                                           (cd (component_td x))
                                           (dv (udv x)))) ))

(prove-lemma listp_array_desc_subtrees (rewrite)
  (implies (array_descp x)
           (listp (subtrees x)))
  ( (enable subtree_body subtree list_subtree) ))

(prove-lemma listp_record_desc_subtrees (rewrite)
  (implies (record_descp x)
           (listp (subtrees x)))
  ( (enable subtree_body subtree list_subtree) ))

(prove-lemma listp_mapping_desc_subtrees (rewrite)
  (implies (mapping_descp x)
           (listp (subtrees x)))
  ( (enable subtree_body subtree list_subtree) ))

(prove-lemma listp_sequence_desc_subtrees (rewrite)
  (implies (sequence_descp x)
           (listp (subtrees x)))
  ( (enable subtree_body subtree list_subtree) ))

(prove-lemma listp_set_desc_subtrees (rewrite)
  (implies (set_descp x)
           (listp (subtrees x)))
  ( (enable subtree_body subtree list_subtree) ))

(prove-lemma treep_type_desc (rewrite)
  (implies (type_descp x)
```

```
           (treep x)))

(prove-lemma lessp_cdr_tree_size (rewrite)
  (implies (listp x)
           (lessp (tree_size (cdr x))
                  (tree_size x))))

(prove-lemma lessp_cdar_tree_size (rewrite)
  (implies (listp x)
           (lessp (tree_size (cdar x))
                  (tree_size x))))



; ======================
;  The Vequal Do-Mutual
; ======================

(disable mk_set_desc)
(disable type_descp)
(disable array_descp)
(disable record_descp)
(disable mapping_descp)
(disable sequence_descp)
(disable set_descp)


(do-mutual '(

(defn vmapped_value (m k td)
  ; m is a selector-component map
  ; k is a Gypsy value, the selector
  ; td is the type descriptor for k and keys of m
  ; result is m[k]
  (if (nlistp m)
      (no_such_component_error m k)
  (if (vequal (caar m) k td)
      (cdar m)
    (vmapped_value (cdr m) k td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
                           (add1 (count m)))
                     (count k))) ))

(defn vmapped_value_list (m ks td)
  ; m is a selector-component-map
  ; ks is a list of Gypsy values, the selectors
  ; td is the type descriptor for elements of ks and keys of m
  ; result is a list of components
  (if (nlistp ks)
      nil
    (cons (vmapped_value m (car ks) td)
          (vmapped_value_list m (cdr ks) td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
                           (add1 (count m)))
                     (count ks))) ))

(defn vequal_list (v1 v2 td)
  (if (nlistp v1)
      (nlistp v2)
  (if (nlistp v2)
      F
    (and (vequal (car v1) (car v2) td)
         (vequal_list (cdr v1) (cdr v2) td))))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
                           (add1 (count v1)))
                     (count v2))) ))

(defn varray_equal (v1 v2 td)
  ; entry (and (array_descp td)
```

```
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  ; Note: Because v1 and v2 are both in type td
  ;       (and (equal (vindexes v1) (value_set (selector_td td)))
  ;            (equal (vindexes v2) (value_set (selector_td td))))
  (if (array_descp td)
      (vequal_list (vcomponents v1)
                   (vmapped_value_list v2 (vindexes v1) (selector_td td))
                   (component_td td))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vfields_equal (v1 v2 tds)
  ; entry (and (equal (field_names v1) (field_names v2))
  ;            (subset (field_names tds) (field_names v1)))
  (if (nlistp tds)
      T
    (and (vequal (fselect v1 (caar tds))
                 (fselect v2 (caar tds))
                 (cdar tds))
         (vfields_equal v1 v2 (cdr tds))))
  ( (ord-lessp (cons (cons (add1 (tree_size tds))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vrecord_equal (v1 v2 td)
  ; entry (and (record_descp td)
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  ; Note: Because v1 and v2 are both in type td
  ;       (and (equal (field_names v1) (field_names (field_tds td)))
  ;            (equal (field_names v2) (field_names (field_tds td))))
  (if (record_descp td)
      (vfields_equal v1 v2 (field_tds td))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vmapping_equal (v1 v2 td)
  ; entry (and (mapping_descp td)
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  (if (mapping_descp td)
      (and (vequal (vdomain v1) (vdomain v2)
                   (mk_set_desc nil (selector_td td) (null_set)))
           (vequal_list (vcomponents v1)
                        (vmapped_value_list v2 (vdomain v1) (selector_td td))
                        (component_td td)))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vsequence_equal (v1 v2 td)
  ; entry (and (sequence_descp td)
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  (if (sequence_descp td)
      (vequal_list v1 v2 (component_td td))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vset_equal (v1 v2 td)
  ; entry (and (set_descp td)
```

```
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  (if (set_descp td)
      (and (vsubsetp v1 v2 (component_td td))
           (vsubsetp v2 v1 (component_td td)))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size (subtrees td)))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vequal (v1 v2 td)
  ; entry (and (type_descp td)
  ;            (truep (dtype td v1))
  ;            (truep (dtype td v2)))
  (if (type_descp td)
      (case (root td)
        (array (varray_equal v1 v2 td))
        (record (vrecord_equal v1 v2 td))
        (mapping (vmapping_equal v1 v2 td))
        (sequence (vsequence_equal v1 v2 td))
        (set (vset_equal v1 v2 td))
        (rational (requal v1 v2))
        (otherwise ; td is a non-rational simple type
          (equal v1 v2)))
    F)
  ( (ord-lessp (cons (cons (add1 (tree_size td))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vsubsetp (v1 v2 td)
  ; td is the type descriptor for v1's and v2's components
  (if (nlistp v1)
      T
    (and (vmember (car v1) v2 td)
         (vsubsetp (cdr v1) v2 td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
                           (add1 (count v1)))
                     (count v2))) ))

(defn vmember (e s td)
  ; td is the type descriptor for e and s's components
  (if (nlistp s)
      F
    (or (vequal e (car s) td)
        (vmember e (cdr s) td)))
  ( (ord-lessp (cons (cons (add1 (tree_size td))
                           (add1 (count e)))
                     (count s))) ))

))


(defn vsize (v)
  (length v))

(defn vseq_select (v k)
  (if (and (numberp k)
           (leq 1 k)
           (leq k (vsize v)))
      (nth k v)
    (no_such_component_error v k)))

(defn vsubseq_select (v lo hi)
  (if (greaterp lo (vsize v))
      nil
  (if (and (leq 1 lo) (leq lo hi))
      (cons (vseq_select v lo)
            (vsubseq_select v (add1 lo) hi))
    nil))
```

```
  ( (lessp (difference hi (sub1 lo))) ))

(defn vselect (v k td)
  ; entry (dtype td v) & k is the right type selector for v
  (if (type_descp td)
      (case (root td)
        (array (vmapped_value v k (selector_td td)))
        (record (fselect v k))
        (mapping (vmapped_value v k (selector_td td)))
        (sequence (vseq_select v k))
        (otherwise (not_selectable_error td)))
    (not_type_descriptor_error td)))

(defn vmap_put (m k v td)
  ; td is the type descriptor for k and m's keys
  (if (nlistp m)
      (cons (map_entry k v) m)
    (if (vequal (caar m) k td)
        (cons (map_entry k v) (cdr m))
      (cons (car m) (vmap_put (cdr m) k v td)))))

(defn varray_put (a i v td)
  ; entry (and (dtype td a)
  ;            (dtype (selector_td td) i)
  ;            (dtype (component_td td) v)
  ;            (array_descp td))
  (vmap_put a i v (selector_td td)))

(defn vrecord_put (r fn v)
  (add_to_map r fn v))

(defn vmapping_put (m i v td)
  ; entry (and (dtype td m)
  ;            (dtype (selector_td td) i)
  ;            (dtype (component_td td) v)
  ;            (mapping_descp td))
  (vmap_put m i v (selector_td td)))

(defn vsequence_put (s i v)
  (if (nlistp s)
      s
    (if (equal i 1)
        (cons v (cdr s))
      (cons (car s) (vsequence_put (cdr s) (sub1 i) v)))))

(defn vsetp (s td)
  ; td is the type descriptor for s's components
  (if (nlistp s)
      T
    (and (not (vmember (car s) (cdr s) td))
         (vsetp (cdr s) td))))

(defn vset (vs td)
  ; td is the type descriptor for vs's components
  (if (nlistp vs)
      nil
  (if (vmember (car vs) (cdr vs) td)
      (vset (cdr vs) td)
    (cons (car vs) (vset (cdr vs) td)))))

(defn vsubmapp (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2) (mapping_descp td))
  (if (nlistp v1)
      T
    (let ((c (vselect v2 (caar v1) td)))
      (if (errorp c)
          F
        (and (vequal (cdar v1) c (component_td td))
             (vsubmapp (cdr v1) v2 td))))))
```

```
(defn vsubseqp (v1 v2 td)
  ; entry (and (dtype (sequence_desc nil td) v1)
  ;            (dtype (sequence_desc nil td) v2))
  (if (nlistp v1)
      T
    (if (nlistp v2)
        F
      (if (vequal (car v1) (car v2) td)
          (vsubseqp (cdr v1) (cdr v2) td)
        (vsubseqp v1 (cdr v2) td)))))

(defn vsubp (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2)
  ;            (or (mapping_descp td) (sequence_descp td) (set_descp td)))
  (case (root td)
    (mapping (vsubmapp v1 v2 td))
    (sequence (vsubseqp v1 v2 (component_td td)))
    (set (vsubsetp v1 v2 (component_td td)))
    (otherwise F)))

(defn vmap_remove (m k td)
  ; entry (and (dtype td m)
  ;            (dtype (selector_td td) k)
  ;            (mapping_descp td))
  (if (nlistp m)
      m
    (if (vequal (caar m) k (selector_td td))
        (cdr m)
      (cons (car m) (vmap_remove (cdr m) k td)))))

(defn vremove (v1 v2 td)
  ; entry (and (dtype td v1) (dtype (set_desc nil td) v2))
  (if (listp v2)
      (if (vequal v1 (car v2) td)
          (vremove v1 (cdr v2) td)
        (cons (car v2)
              (vremove v1 (cdr v2) td)))
    v2))

(defn vdifference (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2) (set_descp td))
  (if (nlistp v1)
      v1
    (if (vmember (car v1) v2 (component_td td))
        (vdifference (cdr v1) v2 td)
      (cons (car v1) (vdifference (cdr v1) v2 td)))))

(defn vdifference_maps (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2) (mapping_descp td))
  (if (nlistp v1)
      v1
    (let ((c (vselect v2 (caar v1) td)))
      (if (errorp c)
          (cons (car v1) (vdifference_maps (cdr v1) v2 td))
      (if (vequal (cdar v1) c (component_td td))
          (vdifference_maps (cdr v1) v2 td)
        (mapping_merge_error v1 v2))))))

(defn vintersect (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2) (set_descp td))
  (if (nlistp v1)
      v1
    (if (vmember (car v1) v2 (component_td td))
        (cons (car v1) (vintersect (cdr v1) v2 td))
      (vintersect (cdr v1) v2 td))))

(defn vintersect_maps (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2) (mapping_descp td))
  (if (nlistp v1)
```

```
      v1
    (let ((c (vselect v2 (caar v1) td)))
      (if (errorp c)
          (vintersect_maps (cdr v1) v2 td)
      (if (vequal (cdar v1) c (component_td td))
          (cons (car v1) (vintersect_maps (cdr v1) v2 td))
        (mapping_merge_error v1 v2))))))

(defn vunion (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2) (set_descp td))
  (if (nlistp v1)
      v2
    (if (vmember (car v1) v2 (component_td td))
        (vunion (cdr v1) v2 td)
      (cons (car v1) (vunion (cdr v1) v2 td)))))

(defn vunion_maps (v1 v2 td)
  ; entry (and (dtype td v1) (dtype td v2) (mapping_descp td))
  (if (nlistp v1)
      v2
    (let ((c (vselect v2 (caar v1) td)))
      (if (errorp c)
          (cons (car v1) (vunion_maps (cdr v1) v2 td))
      (if (vequal (cdar v1) c (component_td td))
          (vunion_maps (cdr v1) v2 td)
        (mapping_merge_error v1 v2))))))



; ********************
;  Value Sets of Types
; ********************

; ====================
;  Value Set Utilities
; ====================

(defn remove_larger (s n)
  (if (nlistp s)
      s
    (if (greaterp (length (car s)) n)
        (remove_larger (cdr s) n)
      (cons (car s) (remove_larger (cdr s) n)))))

(defn list_cons (x y)
  (if (nlistp y)
      nil
    (cons (cons x (car y))
          (list_cons x (cdr y)))))

(defn all_subsets (vs)
  (if (nlistp vs)
      (list nil)
    (append (list_cons (car vs) (all_subsets (cdr vs)))
            (all_subsets (cdr vs)))))

(defn all_seqs_n (rvs avs n)
  (if (zerop n)
      (list nil)
    (if (nlistp rvs)
        nil
      (append (list_cons (car rvs)
                         (all_seqs_n avs avs (sub1 n)))
              (all_seqs_n (cdr rvs) avs n))))
  ( (ord-lessp (cons (add1 n) (count rvs))) ))

(defn all_seqs_le_n (vs n)
  (if (zerop n)
      (list nil)
```

```
    (append (all_seqs_n vs vs n)
            (all_seqs_le_n vs (sub1 n)))))

(defn indexed_value_set (is rcs acs)
  (if (nlistp is)
      (list nil)
    (if (nlistp rcs)
      nil
    (append (list_cons (cons (car is) (car rcs))
                       (indexed_value_set (cdr is) acs acs))
            (indexed_value_set is (cdr rcs) acs))))
  ( (ord-lessp (cons (add1 (count is))
                     (count rcs))) ))

(defn index_set_value_set (iss cs)
  ; iss is a set of index sets
  ; cs is the component set
  (if (nlistp iss)
      nil
    (append (indexed_value_set (car iss) cs cs)
            (index_set_value_set (cdr iss) cs))))


; ======================
;  Value Set Computation
; ======================

(defn simple_value_set (td)
  ; entry td is a correct type descriptor
  (if (bounded_typep td)
      (if (non_rational_simple_typep td)
          (if (errorp (tmin td))
              (tmin td)
          (if (errorp (tmax td))
              (tmax td)
            (number_list (tmin td) (tmax td))))
        (rational_value_set_error td))
    (unbounded_value_set_error td)))

(defn array_value_set (is cs)
  (if (errorp is)
      is
  (if (errorp cs)
      cs
    (indexed_value_set is cs cs))))

(prove-lemma count_cons (rewrite)
  (implies (listp x)
           (equal (count x)
                  (add1 (plus (count (car x)) (count (cdr x)))))))

(prove-lemma zero_count_imp_not_list (rewrite)
  (implies (equal (count x) 0)
           (not (listp x))))

(prove-lemma listp_imp_count_not_zero (rewrite)
  (implies (listp x)
           (not (equal (count x) 0)))
  ( (use (zero_count_imp_not_list)) ))

(disable count_cons)
(disable zero_count_imp_not_list)

(prove-lemma listp_cdar_imp_sub1_count_not_zero (rewrite)
  (implies (listp (cdar x))
           (not (equal (sub1 (count x)) 0))))

(prove-lemma lessp_plus_cdr_caar_cddar (rewrite)
  (implies (and (listp fs) (listp (cdar fs)))
```

```
           (lessp (plus (count (cdr fs))
                        (count (caar fs))
                        (count (cddar fs)))
                  (sub1 (sub1 (count fs))))))

(defn field_list_sets (fs)
  ; fs is ( (f1 . <f1 value set>) (f2 . <f2 value set>)
  ;         ... (fn . <fn value set>) )
  (if (nlistp fs)
      (list nil)
    (let ((fn (caar fs))
          (fvs (cdar fs)))
      (if (nlistp fvs)
          nil
        (append (list_cons (cons fn (car fvs))
                           (field_list_sets (cdr fs)))
                (field_list_sets (cons (cons fn (cdr fvs))
                                       (cdr fs))))))))

(defn record_value_set (fs)
  (if (errorp fs)
      fs
    (field_list_sets fs)))

(defn mapping_value_set (sl ss cs)
  (if (errorp sl)
      sl
  (if (errorp ss)
      ss
  (if (errorp cs)
      cs
    (let ((r (index_set_value_set (all_subsets ss) cs)))
      (if (equal sl nil)
          r
        (remove_larger r sl)))))))

(defn sequence_value_set (sl cs)
  (if (errorp sl)
      sl
  (if (errorp cs)
      cs
  (if (numberp sl)
      (all_seqs_le_n cs sl)
    (unbounded_sequence_value_set_error)))))

(defn set_value_set (sl cs)
  (if (errorp sl)
      sl
  (if (errorp cs)
      cs
  (if (numberp sl)
      (remove_larger (all_subsets cs) sl)
    (all_subsets cs)))))

(do-mutual '( ; the value_set do-mutual

(defn field_value_sets (fds)
  (if (nlistp fds)
      fds
    (let ((fv1 (value_set (cdar fds)))
          (fvs (field_value_sets (cdr fds))))
      (if (errorp fv1)
          fv1
      (if (errorp fvs)
          fvs
        (cons (map_entry (caar fds) fv1) fvs)))))
  ( (lessp (tree_size fds)) ))

(defn value_set (td)
```

```
  ; entry td is a correct type descriptor
  ; returns an errorp or a list of values
  (if (type_descp td)
      (case (root td)
        (array (array_value_set (value_set (selector_td td))
                                (value_set (component_td td))))
        (record (record_value_set (field_value_sets (field_tds td))))
        (mapping (mapping_value_set (max_size td)
                                    (value_set (selector_td td))
                                    (value_set (component_td td))))
        (sequence (sequence_value_set (max_size td)
                                      (value_set (component_td td))))
        (set (set_value_set (max_size td)
                            (value_set (component_td td))))
        (pending (pending_type_value_set_error td))
        (error* td)
        (otherwise ; (simple_descp td)
          (simple_value_set td)))
    (not_type_descriptor_error td))
  ( (lessp (tree_size td)) ))

))


; See below for marked and typed value sets.



; **************
;  Typed Values
; **************

; ------------------------------------------------------------------------
; A typed value is constructed by calling function
;
;     typed (td,u)
;
; where td is a type descriptor and u is a member of td's value set.
; The value of dtype(td,u) = T iff u is a member of td's value set.
; ------------------------------------------------------------------------

(defn dtype_size (n v)
  (if (errorp n)
      n
  (if (numberp n)
      (leq (length v) n)
    ; (equal n nil)
    T)))


(prove-lemma count_keys (rewrite)
  (implies (and (key_value_mapp v)
                (not (lessp (count (keys v)) (count v))))
           (equal (count (keys v)) (count v)))
  ( (use (lessp_keys (m v))) ))

(prove-lemma count_key_values (rewrite)
  (implies (and (key_value_mapp v)
                (not (lessp (count (key_values v)) (count v))))
           (equal (count (key_values v)) (count v)))
  ( (use (lessp_key_values (m v))) ))


(do-mutual '(

(defn dtype_list (td vs)
  (if (nlistp vs)
      (equal vs nil)
    (let ((r (dtype td (car vs))))
```

```
      (if (truep r)
          (dtype_list td (cdr vs))
        r)))
  ( (ord-lessp (cons (add1 (count vs))
                     (tree_size td))) ))

(defn dtype_fields (fds fvs)
  (if (nlistp fvs)
      (equal fvs nil)
    (let ((r (dtype (mapped_value fds (caar fvs)) (cdar fvs))))
      (if (truep r)
          (dtype_fields fds (cdr fvs))
        r)))
  ( (ord-lessp (cons (add1 (count fvs))
                     (tree_size fds))) ))

(defn dtype (td v)
  ; entry td is a correct type descriptor
  ; v is a Gypsy value
  ; returns T, F, or an errorp
  (if (type_descp td)
      (case (root td)
        (array (if (key_value_mapp v)
                   (let ((ivs (value_set (selector_td td)))
                         (r (dtype_list (selector_td td) (vindexes v))))
                     (if (truep r)
                         (if (errorp ivs)
                             ivs
                           (if (and (vsetp (vindexes v) (selector_td td))
                                    (vset_equal ivs (vindexes v)
                                                (mk_set_desc nil
                                                             (selector_td td)
                                                             (null_set))))
                               (dtype_list (component_td td) (vcomponents v))
                             F))
                       r))
                 F))
        (record (if (and (key_value_mapp v)
                         (setp (field_names v))
                         (set_equal (field_names (field_tds td))
                                    (field_names v)))
                    (dtype_fields (field_tds td) v)
                  F))
        (mapping (if (key_value_mapp v)
                     (let ((r (dtype_list (selector_td td) (vdomain v))))
                       (if (truep r)
                           (if (vsetp (vdomain v) (selector_td td))
                               (let ((r (dtype_list (component_td td)
                                                    (vrange v))))
                                 (if (truep r)
                                     (dtype_size (max_size td) v)
                                   r))
                             F)
                         r))
                   F))
        (sequence (let ((r (dtype_list (component_td td) v)))
                    (if (truep r)
                        (dtype_size (max_size td) v)
                      r)))
        (set (if (vsetp v (component_td td))
                 (let ((r (dtype_list (component_td td) v)))
                   (if (truep r)
                       (dtype_size (max_size td) v)
                     r))
               F))
        (pending (pending_in_type_error td))
        (error* td)
        (otherwise ; (simple_descp td)
          (if (errorp (tmin td))
```

```
              (tmin td)
          (if (errorp (tmax td))
              (tmax td)
          (if (equal (tid td) 'rational)
              (and (rationalp v)
                   (if (bounded_typep td)
                       (and (rleq (tmin td) v)
                            (rleq v (tmax td)))
                     T))
            (and (integerp v)
                 (if (bounded_typep td)
                     (and (ileq (tmin td) v)
                          (ileq v (tmax td)))
                   T)))))))
    (not_type_descriptor_error td))
  ( (ord-lessp (cons (add1 (count v))
                     (tree_size td))) ))

))


(add-shell mk_typed nil typedp
           ((type_part (none-of) false)   ; type descriptor
            (value_part (none-of) zero))) ; member of value set of type_part

(defn typed (td u)
  (if (truep (dtype td u))
      (mk_typed td u)
    (mk_typed (integer_desc) 0)))



; ********************
;  Marked Typed Values
; ********************

; ----------------------------------------------------------------------
; Whether a typed value is determinate or indeterminate is indicated by
; marking it with nil for determinate typed values and non-nil for
; indeterminate typed values.
; ----------------------------------------------------------------------

(defn determinate (x)
  (if (markedp x)
      (equal (mark x) nil)
    F))

(defn indeterminate (x)
  (not (determinate x)))

(defn marked_typed (td u)
  (if (truep (dtype td u))
      (marked nil (typed td u))
    (marked (not_in_type_error u td)
            (typed td u))))

(defn marked_typed_list (td us)
  (if (nlistp us)
      us
    (cons (marked_typed td (car us))
          (marked_typed_list td (cdr us)))))



; ***********************************
;  Extraction of Type and Value Parts
; ***********************************

(defn type (x)
```

```
  ; Extracts the type part of a typed value or a marked typed value.
  ; Untyped values have no type part.
  (if (markedp x)
      (type (unmark x))
  (if (typedp x)
      (type_part x)
    F)))

(defn value (x)
  ; Extracts the value part from a Gypsy value, typed value, or
  ; marked typed value.
  (if (markedp x)
      (value (unmark x))
  (if (typedp x)
      (value_part x)
    x)))

(defn values (s)
  ; Extracts the value parts from a list of Gypsy values, typed values, and
  ; marked typed values.
  (if (nlistp s)
      nil
    (cons (value (car s)) (values (cdr s)))))



; *************************
;  Marked Typed Value Sets
; *************************

(defn marked_typed_value_set (td)
  (let ((vs (value_set td)))
    (if (errorp vs)
        vs
      (marked_typed_list td vs))))



; *************************
;  Default Values of Types
; *************************

(defn mk_array_default (id cd)
  ; entry id is a correct type descriptor for an array index type
  ;       cd is a correct type descriptor
  (let ((is (value_set id))
        (cv (udv cd)))
    (if (errorp is)
        is
    (if (errorp cv)
        cv
      (pair_list_map is (ncopies (length is) cv))))))

(defn mk_record_default (fds)
  ; fds is a key_value_map from field names to field type descriptors
  (if (nlistp fds)
      nil
    (let ((fv1 (udv (cdar fds))))
      (if (errorp fv1)
          fv1
        (let ((fvs (mk_record_default (cdr fds))))
          (if (errorp fvs)
              fvs
            (cons (map_entry (caar fds) fv1) fvs)))))))

(defn default_value (td)
  ; Note:  it may not be possible to construct
  ;        [td,u] such that dtype(td,u)
  ; **** Replace body with (typed td (udv td)) if something reasonable gets
```

```
  ;      worked out for type names and field names (and maybe function names).
  (if (and (type_descp td)
           (not (errorp (udv td))))
      (typed td (udv td))
    (typed (integer_desc) 0)))



; ***********
;  Base Type
; ***********

(do-mutual '(

(defn field_base_types (fds)
  (if (nlistp fds)
      fds
    (cons (map_entry (caar fds) (base_type (cdar fds)))
          (field_base_types (cdr fds))))
  ( (lessp (tree_size fds)) ))

(defn base_type (td)
  ; entry td is a correct type descriptor
  (if (type_descp td)
      (case (root td)
        (integer (integer_desc))
        (rational (rational_desc))
        (scalar (mk_scalar_desc (tid td) (sid td) (crd td)
                                0 (sub1 (crd td)) 0))
        (array (let ((id (selector_td td))
                     (cd (base_type (component_td td))))
                 (mk_array_desc id cd (mk_array_default id cd))))
        (record (let ((fds (field_base_types (field_tds td))))
                  (mk_record_desc fds (mk_record_default fds))))
        (mapping (mk_mapping_desc nil (base_type (selector_td td))
                                  (base_type (component_td td))
                                  (null_map)))
        (sequence (mk_sequence_desc nil (base_type (component_td td))
                                    (null_seq)))
        (set (mk_set_desc nil (base_type (component_td td)) (null_set)))
        (pending td)
        (error* td)
        (otherwise (not_type_descriptor_error td)))
    (not_type_descriptor_error td))
  ( (lessp (tree_size td)) ))

))



; ***************
;  Type Equality
; ***************

(defn type_vequal (v1 v2 td)
  (if (or (errorp v1) (errorp v2))
      F
  (if (equal v1 nil)
      (equal v2 nil)
    (vequal v1 v2 td))))

(do-mutual '(

(defn field_tds_equal (t1 t2)
  (if (nlistp t1)
      T
    (and (type_equal (cdar t1) (mapped_value t2 (caar t1)))
         (field_tds_equal (cdr t1) t2)))
  ( (lessp (tree_size t1)) ))
```

```
(defn type_equal (t1 t2)
  (if (and (type_descp t1) (type_descp t2)
           (equal (root t1) (root t2)))
      (case (root t1)
        (integer (and (type_vequal (tmin t1) (tmin t2) t1)
                      (type_vequal (tmax t1) (tmax t2) t1)))
        (rational (and (type_vequal (tmin t1) (tmin t2) t1)
                       (type_vequal (tmax t1) (tmax t2) t1)))
        (scalar (and (equal (tid t1) (tid t2))
                     (equal (sid t1) (sid t2))
                     (type_vequal (crd t1) (crd t2) (integer_desc))
                     (type_vequal (tmin t1) (tmin t2) t1)
                     (type_vequal (tmax t1) (tmax t2) t1)))
        (array (and (type_equal (selector_td t1) (selector_td t2))
                    (type_equal (component_td t1) (component_td t2))))
        (record (and (set_equal (field_names t1) (field_names t2))
                     (field_tds_equal (field_tds t1) (field_tds t2))))
        (mapping (and (type_vequal (max_size t1) (max_size t2) (integer_desc))
                      (type_equal (selector_td t1) (selector_td t2))
                      (type_equal (component_td t1) (component_td t2))))
        (sequence (and (type_vequal (max_size t1) (max_size t2) (integer_desc))
                       (type_equal (component_td t1) (component_td t2))))
        (set (and (type_vequal (max_size t1) (max_size t2) (integer_desc))
                  (type_equal (component_td t1) (component_td t2))))
        (pending (and (equal (tid t1) (tid t2))
                      (equal (sid t1) (sid t2))))
        (otherwise F))
    F)
  ( (lessp (tree_size t1)) ))

))



; *********
;  In Type
; *********

(defn in_type (td u)
  (if (error_descp td)
      td
  (if (error_descp (type u))
      (type u)
  (if (type_equal td (type u))
      T
  (if (equal (base_type td) (base_type (type u)))
      (dtype td (value u))
    F)))))



; ****************************************
;  Lemmas for Proving Type_Desc Terminates
; ****************************************

(defn all_type_units (ul sn)
  (if (nlistp ul)
      nil
    (if (equal (kind (car ul)) 'type)
        (cons (cons sn (car ul))
              (all_type_units (cdr ul) sn))
      (all_type_units (cdr ul) sn))))

(defn all_scope_types (sl)
  (if (nlistp sl)
      nil
    (append (all_type_units (unit_list (car sl))
                            (scope_name (car sl)))
            (all_scope_types (cdr sl)))))
```

```
(defn all_Gypsy_types (x)
  (all_scope_types (scope_list x)))

(prove-lemma all_type_units_members (rewrite)
  (implies (and (member u ul)
                (equal (kind u) 'type))
           (member (cons sn u)
                   (all_type_units ul sn))))

(prove-lemma member_append (rewrite)
  (implies (or (member e x) (member e y))
           (member e (append x y))))

(prove-lemma all_scope_types_members (rewrite)
  (implies (and (member s sl)
                (member u (unit_list s))
                (equal (kind u) 'type))
           (member (cons (scope_name s) u)
                   (all_scope_types sl)))
  ( (disable all_type_units kind scope_name unit_list) ))

(prove-lemma all_Gypsy_type_members (rewrite)
  (implies (and (member s (scope_list x))
                (member u (unit_list s))
                (equal (kind u) 'type))
           (member (cons (scope_name s) u)
                   (all_Gypsy_types x))))

(prove-lemma rule_imp_root_equal_lhs (rewrite)
  (implies (rule tr (prodn lhs rhs))
           (equal (root tr) (untag lhs)))
  ( (disable parse_treep)
    (enable rule) ))

(prove-lemma not_root_equal_lhs_imp_not_rule (rewrite)
  (implies (and (not (equal r nt)) (litatom nt))
           (equal (rule (mk_tree r s) (prodn (tag nt l) rhs))
                  F))
  ( (use (rule_imp_root_equal_lhs (tr (mk_tree r s))
                                  (lhs (tag nt l)))) ))

(prove-lemma mk_error_kind (rewrite)
  (equal (kind (mk_error x)) 'error))

(prove-lemma all_scopes_car (rewrite)
  (implies (equal (length (all_scopes sn sl)) 1)
           (and (equal (scope_name (car (all_scopes sn sl))) sn)
                (member (car (all_scopes sn sl)) sl))))

(prove-lemma all_units_car (rewrite)
  (implies (equal (length (all_units un ul)) 1)
           (and (equal (local_name (car (all_units un ul))) un)
                (member (car (all_units un ul)) ul))))

(prove-lemma mref_result (rewrite)
  (implies (not (equal (kind (ref_unit (mref un sn x n))) 'error))
           (and (member (car (all_scopes (ref_scope (mref un sn x n))
                                         (scope_list x)))
                        (scope_list x))
                (equal (scope_name (car (all_scopes
                                         (ref_scope (mref un sn x n))
                                         (scope_list x))))
                       (ref_scope (mref un sn x n)))
                (member (ref_unit (mref un sn x n))
                        (unit_list (car (all_scopes
                                         (ref_scope (mref un sn x n))
                                         (scope_list x)))))))
  ( (disable all_scopes all_units foreign_name foreign_scope_name kind length
             mk_error reserved_idp scope_list unit_list) ))
```

```
(prove-lemma refed_type (rewrite)
  (implies (equal (kind (ref_unit (ref un sn x))) 'type)
           (and (member (car (all_scopes (ref_scope (ref un sn x))
                                         (scope_list x)))
                        (scope_list x))
                (member (ref_unit (ref un sn x))
                        (unit_list (car (all_scopes (ref_scope (ref un sn x))
                                                    (scope_list x)))))
                (equal (scope_name (car (all_scopes (ref_scope (ref un sn x))
                                                    (scope_list x))))
                       (ref_scope (ref un sn x)))))
  ( (disable all_scopes kind member mref ref_scope ref_unit scope_list
             scope_name unit_list) ))

(prove-lemma type_in_all_types (rewrite)
  (implies (equal (kind (ref_unit (ref un sn x))) 'type)
           (member (ref un sn x) (all_Gypsy_types x)))
  ( (disable all_Gypsy_types ref)
    (use (all_Gypsy_type_members (s (car (all_scopes (ref_scope (ref un sn x))
                                                     (scope_list x))))
                                 (u (ref_unit (ref un sn x))))
         (refed_type)) ))

(prove-lemma set_difference_remove (rewrite)
  (implies (listp y)
           (equal (set_difference x y)
                  (remove (car y) (set_difference x (cdr y))))))

(prove-lemma set_difference_member (rewrite)
  (implies (and (member e x) (not (member e y)))
           (member e (set_difference x y))))

(defn available_types (ut x)
  (set_difference (all_Gypsy_types x) ut))

(prove-lemma lessp_available_types (rewrite)
  (implies (and (equal (kind (ref_unit (ref un sn x))) 'type)
                (not (member (ref un sn x) ut)))
           (lessp (length (available_types (cons (ref un sn x) ut) x))
                  (length (available_types ut x))))
  ( (disable all_Gypsy_types ref ref_unit) ))



; ******************************
;  Creation of Type Descriptors
; ******************************

; ===========================
;  Descriptors for the Types
; ===========================

; -------------
;  Simple Types
; -------------

; Descriptors for standard types are defined above:
;
;     type name       function
;     ---------     --------------
;     boolean       boolean_desc
;     character     character_desc
;     integer       integer_desc
;     rational      rational_desc


(defn subrange_desc (td lo hi)
  (let ((rd (set_tmin (set_tmax td hi) lo)))
    (if (truep (dtype rd (udv td)))
```

```
        rd
      (set_udv rd lo))))

(defn scalar_desc (tn sn sc)
  (mk_scalar_desc tn sn sc 0 (sub1 sc) 0))


; -----------------
;  Structured Types
; -----------------

(defn array_desc (id cd)
  (if (error_descp id)
      id
  (if (error_descp cd)
      cd
  (if (index_typep id)
      (mk_array_desc id cd (mk_array_default id cd))
    (array_index_error id)))))

(defn record_desc (fds)
  (if (error_descp fds)
      fds
  (if (setp (field_names fds))
      (if (some_reserved_idp (field_names fds))
          (field_name_reserved_error fds)
        (mk_record_desc fds (mk_record_default fds)))
    (duplicate_field_names_error fds))))

(defn mapping_desc (sl sd cd)
  (if (error_descp sd)
      sd
  (if (error_descp cd)
      cd
  (if (equality_typep sd)
      (mk_mapping_desc sl sd cd (null_map))
    (mapping_selector_type_error sd)))))

(defn sequence_desc (sl cd)
  (if (error_descp cd)
      cd
    (mk_sequence_desc sl cd (null_seq))))

(defn set_desc (sl cd)
  (if (error_descp cd)
      cd
    (mk_set_desc sl cd (null_set))))

(defn pending_desc (tid sid)
  (mk_pending_desc tid sid
                   (pending_default_value_error tid sid)))



; ==================================================
;  Evaluation of Expressions from Type Declarations
; ==================================================

(defn default_initial_value (d sn x)

  (if (rule d (prodn (tag 'opt_default_initial_value_expression 'v)
                     (list 'COLON_EQUAL (tag 'expression 'e))))
      (precomputable_F (subtree d 'expression) sn x)

  (if (rule d (prodn (tag 'opt_default_initial_value_expression 'v)
                     'empty))
      nil

    (opt_default_value_error d sn))))
```

```
(defn size_limit (r sn x)
  (if (rule r (prodn (tag 'opt_size_limit_restriction 'r)
                     (list 'OPEN_PAREN
                           (tag 'expression 'e)
                           'CLOSE_PAREN)))
      (let ((sl (precomputable_F (subtree r 'expression) sn x)))
        (if (determinate sl)
            (let ((ok (in_type (integer_desc) sl)))
              (if (errorp ok)
                  ok
              (if (and (truep ok) (ileq 0 (value sl)))
                  (value sl)
                (size_limit_error r sn))))
          (if (errorp (mark sl))
              (mark sl)
            (mk_error (mark sl)))))

  (if (rule r (prodn (tag 'opt_size_limit_restriction 'r)
                     'empty))
      nil

    (opt_size_limit_error r sn))))


(defn range_min (r sn x)

  (if (rule r (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (range_min (subtree r 'range_limits) sn x)

  (if (rule r (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (precomputable_F (subtree_i r 'expression 1) sn x)

    (not_range_error r sn)))

  ( (lessp (tree_size r)) ))

(defn range_max (r sn x)

  (if (rule r (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (range_max (subtree r 'range_limits) sn x)

  (if (rule r (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (precomputable_F (subtree_i r 'expression 2) sn x)

    (not_range_error r sn)))

  ( (lessp (tree_size r)) ))



; ==========================
;  Range and Default Setting
; ==========================

(defn value_setting (td v vkind)
  (if (errorp v)
      v
  (if (indeterminate v)
      (mk_error v)
    (let ((r (in_type td (unmark v))))
      (if (truep r)
          (value v)
      (if (errorp r)
```

```
          r
        (mk_error (list vkind (unmark v) 'not 'in 'type td)))))))

(defn range_max_setting (td min max)
  (let ((lo (value_setting td min 'range_minimum))
        (hi (value_setting td max 'range_maximum)))
    (if (errorp hi)
        hi
    (if (errorp lo)
        (mk_error (marked lo (unmark max)))
      hi))))

(defn range_min_setting (td min max)
  (let ((lo (value_setting td min 'range_minimum))
        (hi (value_setting td max 'range_maximum)))
    (if (errorp lo)
        lo
    (if (errorp hi)
        (mk_error (marked hi (unmark min)))
      lo))))

(defn set_range (td min max)
  (if (simple_typep td)
      (let ((lo (range_min_setting td min max))
            (hi (range_max_setting td min max)))
        (if (or (and (integerp lo) (not (ileq lo hi)))
                (and (rationalp lo) (not (rleq lo hi))))
            (empty_type_error (set_tmin (set_tmax td hi) lo))
          (subrange_desc td lo hi)))
  (if (error_descp td)
      td
    (non_simple_subrange_type_error td))))

(defn set_default_value (td dv)
  (if (or (equal dv nil) (error_descp td))
      td
    (let ((v (value_setting td dv 'default_initial_value)))
      (set_udv td v))))


; ======================
;  Checking Scalar Types
; ======================

(disable ref)
(disable ref_unit)
(disable errorp)

(defn scalar_check (svs sn x)
  ; Just check that the scalar names are ok and not duplicated.
  (if (nlistp svs)
      T
    (let ((r (ref (car svs) sn x)))
      (if (errorp (ref_unit r))
          (ref_unit r)
        (scalar_check (cdr svs) sn x)))))

(defn construct_scalar_desc (tn sn svs x)
  ; entry (listp svs)
  (let ((err (scalar_check svs sn x)))
    (if (errorp err)
        err
      (scalar_desc tn sn (length svs)))))



; ===================
;  Function Type_Desc
; ===================
```

```
(disable *1*boolean_desc)
(disable *1*character_desc)
(disable *1*integer_desc)
(disable *1*rational_desc)
(disable array_desc)
(disable available_types)
(disable boolean_desc)
(disable character_desc)
(disable construct_scalar_desc)
(disable default_initial_value)
(disable error_descp)
(disable errorp)
(disable gname)
(disable identifierp)
(disable integer_desc)
(disable kind)
(disable length)
(disable mapping_desc)
(disable ncopies)
(disable not_record_fields_error)
(disable not_type_error)
(disable pair_list_map)
(disable pending_desc)
(disable pending_type_defnp)
(disable range_max)
(disable range_min)
(disable rational_desc)
(disable record_desc)
(disable record_field_names)
(disable ref)
(disable ref_scope)
(disable ref_unit)
(disable scalar_type_defnp)
(disable scalar_value_list)
(disable sequence_desc)
(disable set_default_value)
(disable set_desc)
(disable set_range)
(disable size_limit)
(disable type_defn_cycle_error)
(disable unit_name)


(do-mutual '(


(defn field_descs (s sn ut x)

  (if (rule s (prodn (tag 'fields 'f)
                     (list (tag 'fields 'f2) 'SEMI_COLON
                           (tag 'similar_fields 's))))
      (let ((f1 (field_descs (subtree s 'fields) sn ut x))
            (f2 (field_descs (subtree s 'similar_fields) sn ut x)))
        (if (error_descp f1)
            f1
        (if (error_descp f2)
            f2
          (append f1 f2))))

  (if (rule s (prodn (tag 'fields 'f)
                     (tag 'similar_fields 's)))
      (field_descs (subtree s 'similar_fields) sn ut x)

  (if (rule s (prodn (tag 'similar_fields 's)
                     (list (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ft))))
      (let ((ftd (type_desc (subtree s 'type_specification) sn ut x))
            (fns (record_field_names (subtree s 'identifier_list))))
        (if (error_descp ftd)
```

```
            ftd
          (pair_list_map fns (ncopies (length fns) ftd))))

    (not_record_fields_error s sn))))

  ( (ord-lessp (cons (add1 (length (available_types ut x)))
                     (tree_size s))) ))


(defn type_desc (s sn ut x)
  ; sn is the name of the scope where s is to be interpreted
  ; s is a type specification parse tree or a parse tree that arises in its
  ;   interpretation
  ; ut (used types) is a list of types that have already been looked up in x
  ; x is the Gypsy parse tree

  (if (rule s (prodn (tag 'array_type 'a)
                     (list 'ARRAY 'OPEN_PAREN (tag 'type_specification 'it)
                           'CLOSE_PAREN 'OF (tag 'type_specification 'ct))))
      (array_desc (type_desc (subtree_i s 'type_specification 1) sn ut x)
                  (type_desc (subtree_i s 'type_specification 2) sn ut x))

  (if (rule s (prodn (tag 'mapping_type 'm)
                     (list 'MAPPING (tag 'opt_size_limit_restriction 'r)
                           'FROM (tag 'type_specification 'st)
                           'TO (tag 'type_specification 'ct))))
      (mapping_desc (size_limit (subtree s 'opt_size_limit_restriction) sn x)
                    (type_desc (subtree_i s 'type_specification 1) sn ut x)
                    (type_desc (subtree_i s 'type_specification 2) sn ut x))

  (if (rule s (prodn (tag 'record_type 'r)
                     (list 'RECORD 'OPEN_PAREN (tag 'fields 'f)
                           'CLOSE_PAREN)))
      (record_desc (field_descs (subtree s 'fields) sn ut x))

  (if (rule s (prodn (tag 'sequence_type 's)
                     (list 'SEQUENCE (tag 'opt_size_limit_restriction 'r)
                           'OF (tag 'type_specification 'ct))))
      (sequence_desc (size_limit (subtree s 'opt_size_limit_restriction)
                                 sn x)
                     (type_desc (subtree s 'type_specification) sn ut x))

  (if (rule s (prodn (tag 'set_type 's)
                     (list 'SET (tag 'opt_size_limit_restriction 'r)
                           'OF (tag 'type_specification 'ct))))
      (set_desc (size_limit (subtree s 'opt_size_limit_restriction) sn x)
                (type_desc (subtree s 'type_specification) sn ut x))

  (if (rule s (prodn (tag 'type_declaration 'd)
                     (list 'TYPE (tag 'IDENTIFIER 'tn) 'EQUAL
                           (tag 'type_definition 'd2))))
      (if (pending_type_defnp s)
          (pending_desc (unit_name s) sn)
      (if (scalar_type_defnp s)
          (construct_scalar_desc (unit_name s) sn (scalar_value_list s) x)
        (type_desc (subtree s 'type_definition) sn ut x)))

  (if (rule s (prodn (tag 'type_definition 'd)
                     (tag 'array_type 'a)))
      (type_desc (subtree s 'array_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 'd)
                     (tag 'record_type 'r)))
      (type_desc (subtree s 'record_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 'd)
                     (tag 'mapping_type 'm)))
      (type_desc (subtree s 'mapping_type) sn ut x)
```

```
  (if (rule s (prodn (tag 'type_definition 'd)
                     (tag 'sequence_type 's)))
      (type_desc (subtree s 'sequence_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 'd)
                     (tag 'set_type 's)))
      (type_desc (subtree s 'set_type) sn ut x)

  (if (rule s (prodn (tag 'type_definition 's)
                     (list (tag 'type_specification 's)
                           (tag 'opt_default_initial_value_expression 'v))))
      (set_default_value (type_desc (subtree s 'type_specification) sn ut x)
                         ; set_default_value does not change the default
                         ; value if (default_initial_value ...) is nil
                         (default_initial_value
                           (subtree s 'opt_default_initial_value_expression)
                           sn x))

  (if (rule s (prodn (tag 'type_specification 's)
                     (tag 'IDENTIFIER 'tn)))
      (type_desc (subtree s 'IDENTIFIER) sn ut x)

  (if (rule s (prodn (tag 'type_specification 's)
                     (list (tag 'IDENTIFIER 'tn) (tag 'range 'r))))
      (set_range (type_desc (subtree s 'IDENTIFIER) sn ut x)
                 (range_min (subtree s 'range) sn x)
                 (range_max (subtree s 'range) sn x))

  (if (identifierp s)
      (let ((tn (gname s)))
        (if (equal tn 'boolean)
            (boolean_desc)
        (if (equal tn 'character)
            (character_desc)
        (if (equal tn 'integer)
            (integer_desc)
        (if (equal tn 'rational)
            (rational_desc)
          (let ((r (ref tn sn x)))
          (let ((h (ref_scope r))
                (u (ref_unit r)))
            (if (member r ut)
                (type_defn_cycle_error tn sn)
            (if (equal (kind u) 'type)
                (type_desc u h (cons r ut) x)
            (if (errorp u)
                u
              (not_type_error tn sn))))))))))

    (not_type_error s sn))))))))))))))

  ( (ord-lessp (cons (add1 (length (available_types ut x)))
                     (tree_size s))) ))

))


; ****************
;  Literal Values
; ****************

; ----------------
;  Boolean Values
; ----------------

(defn Gfalse ()
  (marked nil (typed (boolean_desc) 0)))
```

```
(defn Gtrue ()
  (marked nil (typed (boolean_desc) 1)))

(defn Gtruep (v)
  (and (determinate v)
       (truep (in_type (boolean_desc) v))
       (equal (value v) (value (Gtrue)))))


; -----------------
;  Character Values
; -----------------

(defn Gchar (c)
  (if (character_valuep c)
      (marked nil (typed (character_desc) (cadr (lexeme c))))
    (marked (character_error c)
            (default_value (character_desc)))))


; -------------------------------
;  Numbers - Integer and Rational
; -------------------------------

(defn Gizero ()
  (marked nil (typed (integer_desc) 0)))

(defn Grzero ()
  (marked nil (typed (rational_desc) (rational 0 1))))

(defn Gione ()
  (marked nil (typed (integer_desc) 1)))

(defn Gitwo ()
  (marked nil (typed (integer_desc) 2)))

(defn char_digit (c)
  (if (and (leq (ascii_0) c) (leq c (ascii_9)))
      (difference c (ascii_0))
  (if (and (leq (ascii_A) c) (leq c (ascii_F)))
      (plus (difference c (ascii_A)) 10)
  (if (and (leq (ascii_lc_a) c) (leq c (ascii_lc_f)))
      (plus (difference c (ascii_lc_a)) 10)
    nil))))

(defn digit_valid (d b)
  ; d is an ASCII character; b is a numerical base > 0
  (let ((n (char_digit d)))
    (and (numberp n)
         (leq 0 n)
         (lessp n b))))

(disable digit_valid)
(disable char_digit)

(defn digit_value (s b)
  (if (nlistp s)
      0
    (if (equal (digit_value (rcdr s) b) 'ind)
        'ind
      (if (digit_valid (rcar s) b)
          (plus (times (digit_value (rcdr s) b) b) (char_digit (rcar s)))
        'ind))))

(defn tdigit_value (s b)
  (if (digit_listp s)
      (digit_value (lexeme s) b)
    'ind))
```

```
(defn mdigit_value (s b)
  (if (equal (tdigit_value s b) 'ind)
      (marked (number_error s b)
              (default_value (integer_desc)))
    (marked nil (typed (integer_desc) (tdigit_value s b)))))

(defn minteger (i)

  (if (rule i (prodn (tag 'number 'n)
                     (tag 'DIGIT_LIST 's)))
      (mdigit_value (subtree i 'DIGIT_LIST) 10)

  (if (digit_listp i)
      (mdigit_value i 10)

  (if (rule i (prodn (tag 'number 'n)
                     (list (tag 'base 'b) (tag 'DIGIT_LIST 's))))
      (mdigit_value (subtree i 'DIGIT_LIST)
                    (ibase (subtree i 'base)))

    (marked (number_error i 10)
            (default_value (integer_desc))) )))

  ( (lessp (tree_size i)) ))


; --------------
;  String Values
; --------------

(defn string_char_seq (s)
  ; entry: (string_char_listp s)
  (if (nlistp s)
      nil
  (if (equal (car s) (ascii_double_quote))
      (cons (car s) (string_char_seq (cddr s)))
    (cons (car s) (string_char_seq (cdr s))))))

(defn Gstring_seq (s)
  (if (string_valuep s)
      (marked nil
              (typed (sequence_desc nil (character_desc))
                     (string_char_seq (rcdr (cdr (lexeme s))))))
    (marked (bad_string_error s)
            (default_value (sequence_desc nil (character_desc))))))



; ***************************************
;  Arguments to Functions and Operations
; ***************************************

(defn in_arg_type (td a)
  (if (equal td 'field_name)
      (and (equal (mark a) td)
           (litatom (unmark a)))
  (if (equal td 'type_descriptor)
      (and (equal (mark a) td)
           (type_descp (unmark a)))
    (in_type td a))))

(defn arg_check (as ts)
  ; as is the list of actual parameters, a list of marked objects
  ;    (typed values, record field names, or type descriptors for type name
  ;    arguments to standard functions)
  ; ts is the list of formal types, a list of type descriptors or occurrences
  ;    of 'field_name or 'type_descriptor
  ; returns nil if the args are ok, otherwise an error
  (if (nlistp as)
```

```
      (if (nlistp ts)
          nil
        (mk_error '(Number of formal and actual parameters not equal)))
  (if (nlistp ts)
      (mk_error '(Number of formal and actual parameters not equal))
  (if (indeterminate (car as))
      (mark (car as))
    (let ((r (in_arg_type (car ts) (car as))))
      (if (truep r)
          (arg_check (cdr as) (cdr ts))
      (if (errorp r)
          r
        (actual_formal_type_error (car as) (car ts)))))))))



; *****************
;  Gypsy Operations
; *****************

; --------------------
;  Component Selection
; --------------------

(defn array_get (a i td)
  (let ((arg_err (arg_check (list a i)
                            (list td (selector_td td)))))
    (if (equal arg_err nil)
        (if (array_descp td)
            (marked nil (typed (component_td td)
                               (vselect (value a) (value i) td)))
          (marked (not_array_error a)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn mapping_get (m d td)
  (let ((arg_err (arg_check (list m d) (list td (selector_td td)))))
    (if (equal arg_err nil)
        (if (mapping_descp td)
            (let ((v (vselect (value m) (value d) td)))
              (if (errorp v)
                  (marked (no_such_component_error m d)
                          (default_value (component_td td)))
                (marked nil (typed (component_td td) v))))
          (marked (not_mapping_error m)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn record_get (r fn td)
  (let ((arg_err (arg_check (list r fn) (list td 'field_name))))
    (if (equal arg_err nil)
        (if (record_descp td)
            (let ((fd (field_td (unmark fn) td)))
              (if (equal fd nil)
                  (marked (no_such_field_error r (unmark fn))
                          (default_value (integer_desc)))
                (marked nil (typed fd (fselect (value r) (unmark fn))))))
          (marked (not_record_error r)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn sequence_get (s i td)
  (let ((arg_err (arg_check (list s i) (list td (integer_desc)))))
    (if (equal arg_err nil)
        (if (sequence_descp td)
            (let ((v (vselect (value s) (value i) td)))
              (if (errorp v)
                  (marked (no_such_component_error s i)
                          (default_value (component_td td)))
```

```
                (marked nil (typed (component_td td) v))))
          (marked (not_sequence_error s)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(disable type)
(disable array_get)
(disable record_get)
(disable mapping_get)
(disable sequence_get)
(disable not_selectable_error)
(disable default_value)

(defn select_op (v s)
  (if (nlistp s)
      v
    (case (root (type v))
      (array (select_op (array_get v (car s) (type v))
                        (cdr s)))
      (record (select_op (record_get v (car s) (type v))
                         (cdr s)))
      (mapping (select_op (mapping_get v (car s) (type v))
                          (cdr s)))
      (sequence (select_op (sequence_get v (car s) (type v))
                           (cdr s)))
      (otherwise (marked (not_selectable_error v)
                         (default_value (integer_desc)))))))



; ---------------------------
;  Sequence/Set Constructors
; ---------------------------

(defn Gseq (es td)
  (let ((arg_err (arg_check es (ncopies (length es) td))))
    (if (equal arg_err nil)
        (marked nil (typed (sequence_desc nil td) (values es)))
      (marked arg_err (default_value (integer_desc))))))

(defn Gset (es td)
  (let ((arg_err (arg_check es (ncopies (length es) td))))
    (if (equal arg_err nil)
        (marked nil (typed (set_desc nil td) (vset (values es) td)))
      (marked arg_err (default_value (integer_desc))))))

(defn Grange_elements (lo hi)
  (let ((arg_err (arg_check (list lo hi)
                            (list (base_type (type lo))
                                  (base_type (type lo))))))
    (if (equal arg_err nil)
        (if (non_rational_simple_typep (type lo))
            (marked_typed_list (base_type (type lo))
                               (number_list (value lo) (value hi)))
          (list (marked (range_limits_error lo hi)
                        (default_value (base_type (type lo))))))
      (list (marked arg_err
                    (default_value (base_type (type lo))))))))

(defn Gset_or_seq (m a td)

  (if (rule m (prodn (tag 'set_or_seq_mark 'm)
                     (list 'SET 'COLON)))
      (Gset a td)

  (if (rule m (prodn (tag 'set_or_seq_mark 'm)
                     (list 'SEQ 'COLON)))
      (Gseq a td)
```

```
    (Gseq a td))))


; -----------------------------------
;  Interface to Boyer-Moore Functions
; -----------------------------------

(defn T_or_F (x)
  (if (zerop x)
      F
    T))

(defn Band (x y)
  (and (T_or_F x) (T_or_F y)))

(defn Bimp (x y)
  (implies (T_or_F x) (T_or_F y)))

(defn Bnot (x)
  (not (T_or_F x)))

(defn Bor (x y)
  (or (T_or_F x) (T_or_F y)))


; ----------------
;  Unary Operators
; ----------------

(defn Gminus (u)
  (let ((iarg_err (arg_check (list u) (list (integer_desc))))
        (rarg_err (arg_check (list u) (list (rational_desc)))))
    (if (equal iarg_err nil)
        (marked nil (typed (integer_desc) (ineg (value u))))
    (if (equal rarg_err nil)
        (marked nil (typed (rational_desc) (rneg (value u))))
      (marked (mk_error (list iarg_err rarg_err))
              (default_value (integer_desc)))))))

(defn Gnot (u)
  (let ((arg_err (arg_check (list u) (list (boolean_desc)))))
    (if (equal arg_err nil)
        (if (Bnot (value u))
            (Gtrue)
          (Gfalse))
      (marked arg_err (default_value (boolean_desc))))))

(defn apply_unary_op (op v)
  (if (rule op (prodn (tag 'unary_operator 'op) 'MINUS))
      (Gminus v)

  (if (rule op (prodn (tag 'unary_operator 'op) 'NOT))
      (Gnot v)

    (marked (not_unary_op_error op)
            (default_value (integer_desc))))))


; -----------------
;  Binary Operators
; -----------------

(defn Gequal (v1 v2)
  (if (equality_typep (type v1))
      (let ((td (base_type (type v1))))
      (let ((arg_err (arg_check (list v1 v2) (list td td))))
        (if (equal arg_err nil)
            (if (vequal (value v1) (value v2) td)
```

```
                (Gtrue)
              (Gfalse))
          (marked arg_err (default_value (boolean_desc))))))
    (marked (not_equality_type_error (type v1))
            (default_value (boolean_desc)))))

(defn Gne (v1 v2)
  (Gnot (Gequal v1 v2)))

(defn Glt (v1 v2)
  (let ((td (base_type (type v1))))
  (let ((arg_err (arg_check (list v1 v2) (list td td))))
    (if (equal arg_err nil)
        (if (simple_typep td)
            (let ((r (if (equal (root td) 'rational)
                         (rlessp (value v1) (value v2))
                       (ilessp (value v1) (value v2)))))
              (if (truep r) (Gtrue) (Gfalse)))
          (marked (not_defined_on_type_error 'LT (type v1))
                  (default_value (boolean_desc))))
      (marked arg_err (default_value (boolean_desc)))))))

(defn Gor (v1 v2)
  (let ((arg_err (arg_check (list v1 v2)
                            (list (boolean_desc) (boolean_desc)))))
    (if (equal arg_err nil)
        (if (Bor (value v1) (value v2))
            (Gtrue)
          (Gfalse))
      (marked arg_err (default_value (boolean_desc))))))

(defn Gle (v1 v2)
  (Gor (Glt v1 v2)
       (Gequal v1 v2)))

(defn Ggt (v1 v2)
  (Glt v2 v1))

(defn Gge (v1 v2)
  (Gor (Ggt v1 v2)
       (Gequal v1 v2)))

(defn Gand (v1 v2)
  (let ((arg_err (arg_check (list v1 v2)
                            (list (boolean_desc) (boolean_desc)))))
    (if (equal arg_err nil)
        (if (Band (value v1) (value v2))
            (Gtrue)
          (Gfalse))
      (marked arg_err (default_value (boolean_desc))))))

(defn Gimp (v1 v2)
  (let ((arg_err (arg_check (list v1 v2)
                            (list (boolean_desc) (boolean_desc)))))
    (if (equal arg_err nil)
        (if (Bimp (value v1) (value v2))
            (Gtrue)
          (Gfalse))
      (marked arg_err (default_value (boolean_desc))))))

(defn Giff (v1 v2)
  (let ((arg_err (arg_check (list v1 v2)
                            (list (boolean_desc) (boolean_desc)))))
    (if (equal arg_err nil)
        (Gequal v1 v2)
      (marked  arg_err (default_value (boolean_desc))))))

(defn Gpower (v1 v2) ; v1 ** v2
  (let ((iarg_err (arg_check (list v1 v2)
```

```
                             (list (integer_desc) (integer_desc))))
        (rarg_err (arg_check (list v1 v2)
                             (list (rational_desc) (integer_desc)))))
    (if (equal iarg_err nil)
        (if (Gtruep (Glt v2 (Gizero)))
            (marked (negative_exponent_error)
                    (default_value (integer_desc)))
        (if (Gtruep (Gand (Gequal v1 (Gizero))
                          (Gequal v2 (Gizero))))
            (marked (zero_to_the_zero_power_error)
                    (default_value (integer_desc)))
          (marked nil (typed (integer_desc) (ipower (value v1) (value v2))))))
    (if (equal rarg_err nil)
        (if (Gtruep (Glt v2 (Gizero)))
            (marked (negative_exponent_error)
                    (default_value (rational_desc)))
        (if (Gtruep (Gand (Gequal v1 (Grzero))
                          (Gequal v2 (Gizero))))
            (marked (zero_to_the_zero_power_error)
                    (default_value (rational_desc)))
          (marked nil (typed (rational_desc) (rpower (value v1) (value v2))))))
      (marked (mk_error (list iarg_err rarg_err))
              (default_value (integer_desc)))))))

(defn Gtimes (v1 v2)
  (let ((iarg_err (arg_check (list v1 v2)
                             (list (integer_desc) (integer_desc))))
        (rarg_err (arg_check (list v1 v2)
                             (list (rational_desc) (rational_desc)))))
    (if (equal iarg_err nil)
        (marked nil (typed (integer_desc) (itimes (value v1) (value v2))))
    (if (equal rarg_err nil)
        (marked nil (typed (rational_desc) (rtimes (value v1) (value v2))))
      (marked (mk_error (list iarg_err rarg_err))
              (default_value (integer_desc)))))))

(defn Gquotient (v1 v2)
  (let ((iarg_err (arg_check (list v1 v2)
                             (list (integer_desc) (integer_desc))))
        (rarg_err (arg_check (list v1 v2)
                             (list (rational_desc) (rational_desc)))))
    (if (equal iarg_err nil)
        (if (izerop (value v2))
            (marked (zero_divide_error)
                    (default_value (rational_desc)))
          (marked nil
                  (typed (rational_desc)
                         (reduce (rational (value v1) (value v2))))))
    (if (equal rarg_err nil)
        (if (rzerop (value v2))
            (marked (zero_divide_error)
                    (default_value (rational_desc)))
          (marked nil (typed (rational_desc)
                             (rquotient (value v1) (value v2)))))
      (marked (mk_error (list iarg_err rarg_err))
              (default_value (rational_desc)))))))

(defn Gdiv (v1 v2)
  (let ((arg_err (arg_check (list v1 v2)
                            (list (integer_desc) (integer_desc)))))
    (if (equal arg_err nil)
        (if (izerop (value v2))
            (marked (zero_divide_error)
                    (default_value (integer_desc)))
          (marked nil (typed (integer_desc)
                             (iquotient (value v1) (value v2)))))
      (marked arg_err
              (default_value (integer_desc))))))
```

```
(defn Gmod (v1 v2)
  (let ((arg_err (arg_check (list v1 v2)
                            (list (integer_desc) (integer_desc)))))
    (if (equal arg_err nil)
        (if (izerop (value v2))
            (marked (zero_divide_error)
                    (default_value (integer_desc)))
          (marked nil (typed (integer_desc)
                             (iremainder (value v1) (value v2)))))
      (marked arg_err
              (default_value (integer_desc))))))

(defn Gplus (v1 v2)
  (let ((iarg_err (arg_check (list v1 v2)
                             (list (integer_desc) (integer_desc))))
        (rarg_err (arg_check (list v1 v2)
                             (list (rational_desc) (rational_desc)))))
    (if (equal iarg_err nil)
        (marked nil
                (typed (integer_desc) (iplus (value v1) (value v2))))
    (if (equal rarg_err nil)
        (marked nil
                (typed (rational_desc) (rplus (value v1) (value v2))))
      (marked (mk_error (list iarg_err rarg_err))
              (default_value (integer_desc)))))))

(defn Gsubtract (v1 v2)
  (let ((iarg_err (arg_check (list v1 v2)
                             (list (integer_desc) (integer_desc))))
        (rarg_err (arg_check (list v1 v2)
                             (list (rational_desc) (rational_desc)))))
    (if (equal iarg_err nil)
        (marked nil
                (typed (integer_desc) (idifference (value v1) (value v2))))
    (if (equal rarg_err nil)
        (marked nil
                (typed (rational_desc) (rdifference (value v1) (value v2))))
      (marked (mk_error (list iarg_err rarg_err))
              (default_value (integer_desc)))))))

(defn Gin (v1 v2)
  (let ((td (base_type (type v2))))
    (if (or (sequence_descp td) (set_descp td))
        (let ((arg_err (arg_check (list v1 v2)
                                  (list (component_td td) td))))
          (if (equal arg_err nil)
              (if (vmember (value v1) (value v2) (component_td td))
                  (Gtrue)
                (Gfalse))
            (marked arg_err (default_value (boolean_desc)))))
      (marked (in_arg_error v2)
              (default_value (boolean_desc))))))

(defn mapping_merge_arg_check2 (ks v1 v2)
  (if (nlistp ks)
      T
    (let ((c1 (mapping_get v1 (car ks) (base_type (type v1))))
          (c2 (mapping_get v2 (car ks) (base_type (type v2)))))
      (if (and (determinate c1) (determinate c2))
          (and (Gtruep (Gequal c1 c2))
               (mapping_merge_arg_check2 (cdr ks) v1 v2))
        (mapping_merge_arg_check2 (cdr ks) v1 v2)))))

(defn mapping_merge_arg_check (v1 v2)
  (if (and (mapping_descp (type v1))
           (mapping_descp (type v2)))
      (let ((ks (marked_typed_list (selector_td (base_type (type v1)))
                                   (vdomain (value v1)))))
        (mapping_merge_arg_check2 ks v1 v2))
```

```
    F))

(defn Gunion (v1 v2)
  (let ((td (base_type (type v1))))
  (let ((arg_err (arg_check (list v1 v2) (list td td))))
    (if (equal arg_err nil)
        (if (set_descp td)
            (marked nil (typed td (vunion (value v1) (value v2) td)))
        (if (mapping_descp td)
            (if (mapping_merge_arg_check v1 v2)
                (marked nil (typed td (vunion_maps (value v1) (value v2) td)))
              (marked (mapping_merge_error v1 v2) (default_value td)))
          (marked (union_arg_error v1 v2)
                  (default_value (integer_desc)))))
      (marked arg_err (default_value (integer_desc)))))))

(defn Gadjoin (v1 v2)
  (if (set_descp (type v1))
      (Gunion v1 (Gset (list v2) (base_type (type v2))))
    (marked (adjoin_args_error v1)
            (default_value (integer_desc)))))

(defn Gomit (v1 v2)
  (let ((td (base_type (type v1))))
    (if (set_descp td)
        (let ((arg_err (arg_check (list v1 v2)
                                  (list td (component_td td)))))
          (if (equal arg_err nil)
              (if (Gtruep (Gin v2 v1))
                  (marked nil
                          (typed td
                                 (vremove (value v2) (value v1)
                                          (component_td td))))
                (marked (not_in_set_error v2 v1)
                        (default_value td)))
            (marked arg_err (default_value (integer_desc)))))
      (marked (omit_args_error v1) (default_value (integer_desc))))))

(defn Gsub (v1 v2)
  (let ((td (base_type (type v1))))
  (let ((arg_err (arg_check (list v1 v2) (list td td))))
    (if (equal arg_err nil)
        (if (or (mapping_descp td) (sequence_descp td) (set_descp td))
            (if (vsubp (value v1) (value v2) td)
                (Gtrue)
              (Gfalse))
          (marked (sub_args_error v1 v2)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn Gintersect (v1 v2)
  (let ((td (base_type (type v1))))
  (let ((arg_err (arg_check (list v1 v2) (list td td))))
    (if (equal arg_err nil)
        (if (set_descp td)
            (marked nil (typed td
                               (vintersect (value v1) (value v2) td)))
        (if (mapping_descp td)
            (if (mapping_merge_arg_check v1 v2)
                (marked nil (typed td
                                   (vintersect_maps (value v1) (value v2) td)))
              (marked (mapping_merge_error v1 v2) (default_value td)))
          (marked (intersect_args_error v1 v2)
                  (default_value (integer_desc)))))
      (marked arg_err (default_value (integer_desc)))))))

(defn Gdifference (v1 v2)
  (let ((td (base_type (type v1))))
  (let ((arg_err (arg_check (list v1 v2) (list td td))))
```

```
    (if (equal arg_err nil)
        (if (set_descp td)
            (marked nil (typed td
                               (vdifference (value v1) (value v2) td)))
        (if (mapping_descp td)
            (if (mapping_merge_arg_check v1 v2)
                (marked nil
                        (typed td
                               (vdifference_maps (value v1) (value v2) td)))
              (marked (mapping_merge_error v1 v2) (default_value td)))
          (marked (difference_args_error v1 v2)
                  (default_value (integer_desc)))))
      (marked arg_err (default_value (integer_desc)))))))

(defn Gappend (v1 v2)
  (let ((td (base_type (type v1))))
  (let ((arg_err (arg_check (list v1 v2) (list td td))))
    (if (equal arg_err nil)
        (if (sequence_descp td)
            (marked nil (typed td
                               (append (value v1) (value v2))))
          (marked (append_args_error v1 v2)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn Gcons (v1 v2) ; v1 :> v2
  (Gappend (Gseq (list v1) (base_type (type v1))) v2))

(defn Grcons (v1 v2) ; v1 <: v2
  (Gappend v1 (Gseq (list v2) (base_type (type v2)))))

(defn apply_binary_op (op v1 v2)

  (if (or (rule op (prodn (tag 'binary_operator 'op) 'EQ))
          (rule op (prodn (tag 'binary_operator 'op) 'EQUAL)))
      (Gequal v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'NE))
      (Gne v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'LT))
      (Glt v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'LE))
      (Gle v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'GT))
      (Ggt v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'GE))
      (Gge v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'AND))
      (Gand v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'OR))
      (Gor v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'IMP))
      (Gimp v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'IFF))
      (Giff v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'STAR_STAR))
      (Gpower v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'STAR))
      (Gtimes v1 v2)
```

```
  (if (rule op (prodn (tag 'binary_operator 'op) 'SLASH))
      (Gquotient v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'DIV))
      (Gdiv v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'MOD))
      (Gmod v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'PLUS))
      (Gplus v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'MINUS))
      (Gsubtract v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'IN))
      (Gin v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'ADJOIN))
      (Gadjoin v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'OMIT))
      (Gomit v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'SUB))
      (Gsub v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'UNION))
      (Gunion v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'INTERSECT))
      (Gintersect v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'DIFFERENCE))
      (Gdifference v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'COLON_GT))
      (Gcons v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'LT_COLON))
      (Grcons v1 v2)

  (if (rule op (prodn (tag 'binary_operator 'op) 'APPEND))
      (Gappend v1 v2)

    (marked (not_binary_op_error op)
            (default_value (integer_desc)))))))))))))))))))))))))))))



; ----------------------
;  Subsequence Selection
; ----------------------

(defn subsequence_get (s lo hi)
  (let ((arg_err (arg_check (list s lo hi)
                            (list (type s) (integer_desc) (integer_desc)))))
    (if (equal arg_err nil)
        (if (sequence_descp (type s))
            (if (Gtruep (Gle lo hi))
                (if (indeterminate (sequence_get s lo (type s)))
                    (marked (no_such_component_error s lo)
                            (default_value (type s)))
                (if (indeterminate (sequence_get s hi (type s)))
                    (marked (no_such_component_error s hi)
                            (default_value (type s)))
                  (marked nil
                          (typed (type s)
                                 (vsubseq_select (value s)
```

```
                                                 (value lo) (value hi))))))
              ; Gypsy 2.05 report says both that the result type of
              ; subsequence_get is type(s) and that the following type should
              ; be base_type(type(s))
              (marked nil (typed (type s) (null_seq))))
          (marked (not_sequence_error s)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))



; -----------------
;  Value Alteration
; -----------------

(defn array_put (a i v)
  (let ((td (base_type (type a))))
  (let ((arg_err (arg_check (list a i v)
                            (list td (selector_td td) (component_td td)))))
    (if (equal arg_err nil)
        (if (array_descp td)
            (marked nil (typed td
                               (varray_put (value a) (value i) (value v) td)))
          (marked (not_array_error a)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn record_put (r fn v)
  (let ((td (base_type (type r))))
  (let ((arg_err (arg_check (list r fn v)
                            (list td 'field_name (field_td (unmark fn) td)))))
    (if (equal arg_err nil)
        (if (record_descp td)
            (marked nil (typed td
                               (vrecord_put (value r) (unmark fn) (value v))))
          (marked (not_record_error r)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn mapping_put (m d v)
  (let ((td (base_type (type m))))
  (let ((arg_err (arg_check (list m d v)
                            (list td (selector_td td) (component_td td)))))
    (if (equal arg_err nil)
        (if (mapping_descp td)
            (if (determinate (mapping_get m d (type m)))
                (marked nil
                        (typed td
                               (vmapping_put (value m) (value d) (value v)
                                             td)))
              (marked (no_such_component_error m d) (default_value td)))
          (marked (not_mapping_error m)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn sequence_put (s i v)
  (let ((td (base_type (type s))))
  (let ((arg_err (arg_check (list s i v)
                            (list td (integer_desc) (component_td td)))))
    (if (equal arg_err nil)
        (if (sequence_descp td)
            (if (determinate (sequence_get s i (type s)))
                (marked nil
                        (typed td
                               (vsequence_put (value s) (value i) (value v))))
              (marked (no_such_component_error s i)
                      (default_value td)))
          (marked (not_sequence_error s)
                  (default_value (integer_desc))))
```

```
      (marked arg_err (default_value (integer_desc)))))))

(defn put_op (bv s v)
  ; bv is the base marked typed value,
  ; s is a list of marked typed values representing component selectors,
  ; v is the new value for the selected component
  ; Example Gypsy:  z with ([i,j,k].f1[l][m].f2 := e)
  (if (nlistp s)
      v
    (case (root (type bv))
      (array (array_put bv (car s)
                        (put_op (array_get bv (car s) (type bv))
                                (cdr s) v)))
      (record (record_put bv (car s)
                          (put_op (record_get bv (car s) (type bv))
                                  (cdr s) v)))
      (mapping (mapping_put bv (car s)
                            (put_op (mapping_get bv (car s) (type bv))
                                    (cdr s) v)))
      (sequence (sequence_put bv (car s)
                              (put_op (sequence_get bv (car s) (type bv))
                                      (cdr s) v)))
      (otherwise (marked (component_assign_error bv)
                         (default_value (integer_desc)))))))

(defn Gmapomit (m i)
  ; m is the base marked typed value,
  ; i is a marked typed value representing a component selector,
  (if (determinate (mapping_get m i (type m)))
      (marked nil (typed (type m)
                         (vmap_remove (value m) (value i) (type m))))
    (marked (mark (mapping_get m i (type m)))
            (default_value (type m)))))

(defn Gseqomit (s i)
  ; s is the base marked typed value,
  ; i is a marked typed value representing a component selector,
  (if (determinate (sequence_get s i (type s)))
      (let ((s2
             (Gappend
               (subsequence_get s (Gione) (Gsubtract i (Gione)))
               (subsequence_get s (Gplus i (Gione))
                                 (marked nil (typed (integer_desc)
                                                    (vsize (value s))))))))
        ; The Gypsy 2.05 report retypes the result of the append
        ; to the type of s.
        (marked (mark s2) (typed (type s) (value s2))))
    (marked (mark (sequence_get s i (type s)))
            (default_value (type s)))))

(defn Gmap_insert (m d v)
  (let ((td (base_type (type m))))
  (let ((arg_err (arg_check (list m d v)
                            (list td (selector_td td) (component_td td)))))
    (if (equal arg_err nil)
        (if (mapping_descp td)
            (if (determinate (mapping_get m d (type m)))
                (mapping_put m d v)
              (marked nil
                      (typed td
                             (vmapping_put (value m) (value d) (value v) td))))
          (marked (not_mapping_error m)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn Gseq_insert_before (s i v)
  (if (determinate (sequence_get s i (type s)))
      (Gappend (subsequence_get s (Gione) (Gsubtract i (Gione)))
               (Gappend
```

```
                  (Gseq (list v) (base_type (type v)))
                  (subsequence_get s i (marked nil
                                             (typed (integer_desc)
                                                    (vsize (value s)))))))
    (marked (mark (sequence_get s i (type s)))
            (default_value (base_type (type s))))))

(defn Gseq_insert_behind (s i v)
  (if (determinate (sequence_get s i (type s)))
      (Gappend (subsequence_get s (Gione) i)
               (Gappend
                 (Gseq (list v) (base_type (type v)))
                 (subsequence_get s (Gplus i (Gione))
                                  (marked nil
                                          (typed (integer_desc)
                                                 (vsize (value s)))))))
    (marked (mark (sequence_get s i (type s)))
            (default_value (base_type (type s))))))



; ********************
;  Standard Functions
; ********************

(defn std_domain (d)
  (let ((arg_err (arg_check d (list (type (car d))))))
    (if (equal arg_err nil)
        (if (mapping_descp (type (car d)))
            (marked nil
                    (typed (set_desc nil (selector_td (type (car d))))
                           (vdomain (value (car d)))))
          (marked (domain_arg_error)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_first (d)
  (let ((arg_err (arg_check d (list (type (car d))))))
    (if (equal arg_err nil)
        (if (sequence_descp (type (car d)))
            (sequence_get (car d) (Gione) (type (car d)))
          (marked (first_arg_error)
                  (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_initial (d)
  (let ((arg_err (arg_check d (list 'type_descriptor))))
    (if (equal arg_err nil)
        (let ((td (unmark (car d))))
          (if (errorp (udv td))
              (marked (udv td) (default_value td))
            (marked nil (typed td (udv td)))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_last (d)
  (let ((arg_err (arg_check d (list (type (car d))))))
    (if (equal arg_err nil)
        (if (sequence_descp (type (car d)))
            (sequence_get (car d)
                          (marked nil (typed (integer_desc) (vsize (car d))))
                          (type (car d)))
          (marked (last_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_lower (d)
  (let ((arg_err (arg_check d (list 'type_descriptor))))
    (if (equal arg_err nil)
        (let ((td (unmark (car d))))
        (let ((r (tmin td)))
```

```
          (if (equal r nil)
              (marked (unbounded_type_error td) (default_value td))
          (if (errorp r)
              (marked r (default_value td))
            (marked nil (typed td r))))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_max (d)
  (let ((td (base_type (type (car d)))))
  (let ((arg_err (arg_check d (list td td))))
    (if (equal arg_err nil)
        (if (simple_typep td)
            (if (Gtruep (Ggt (car d) (cadr d)))
                (marked nil (typed td (value (car d))))
              (marked nil (typed td (value (cadr d)))))
          (marked (max_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn std_min (d)
  (let ((td (base_type (type (car d)))))
  (let ((arg_err (arg_check d (list td td))))
    (if (equal arg_err nil)
        (if (simple_typep td)
            (if (Gtruep (Glt (car d) (cadr d)))
                (marked nil (typed td (value (car d))))
              (marked nil (typed td (value (cadr d)))))
          (marked (min_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn std_nonfirst (d)
  (let ((arg_err (arg_check d (list (type (car d))))))
    (if (equal arg_err nil)
        (if (sequence_descp (type (car d)))
            (if (equal 0 (vsize (value (car d))))
                (marked (empty_seq_error 'nonfirst (car d))
                        (default_value (type (car d))))
              (subsequence_get (car d) (Gitwo)
                               (marked nil
                                       (typed (integer_desc)
                                              (vsize (value (car d)))))))
          (marked (nonfirst_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_nonlast (d)
  (let ((arg_err (arg_check d (list (type (car d))))))
    (if (equal arg_err nil)
        (if (sequence_descp (type (car d)))
            (if (equal 0 (vsize (value (car d))))
                (marked (empty_seq_error 'nonlast (car d))
                        (default_value (type (car d))))
              (subsequence_get (car d) (Gione)
                               (marked nil
                                       (typed (integer_desc)
                                              (sub1
                                                (vsize (value (car d))))))))
          (marked (nonlast_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn Gnull_map (td)
  (if (mapping_descp td)
      (let ((r (dtype td (null_map))))
        (if (truep r)
            (marked nil (typed td (null_map)))
        (if (errorp r)
            (marked r (default_value td))
          (marked (not_in_type_error (null_map) td)
                  (default_value td)))))
    (marked (not_mapping_type_error td) (default_value (integer_desc)))))
```

```
(defn Gnull_seq (td)
  (if (sequence_descp td)
      (let ((r (dtype td (null_seq))))
        (if (truep r)
            (marked nil (typed td (null_seq)))
        (if (errorp r)
            (marked r (default_value td))
          (marked (not_in_type_error (null_seq) td)
                  (default_value td)))))
    (marked (not_sequence_type_error td) (default_value (integer_desc)))))

(defn Gnull_set (td)
  (if (set_descp td)
      (let ((r (dtype td (null_set))))
        (if (truep r)
            (marked nil (typed td (null_set)))
        (if (errorp r)
            (marked r (default_value td))
          (marked (not_in_type_error (null_set) td)
                  (default_value td)))))
    (marked (not_set_type_error td) (default_value (integer_desc)))))

(defn std_null (d)
  (let ((arg_err (arg_check d (list 'type_descriptor))))
    (if (equal arg_err nil)
        (let ((td (unmark (car d))))
          (case (root td)
            (mapping (Gnull_map td))
            (sequence (Gnull_seq td))
            (set (Gnull_set td))
            (otherwise (marked (null_undefined_error td)
                               (default_value (integer_desc))))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_ord (d)
  (let ((td (type (car d))))
  (let ((arg_err (arg_check d (list td))))
    (if (equal arg_err nil)
        (if (scalar_typep td)
            (marked nil (typed (integer_desc) (value (car d))))
          (marked (ord_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn std_pred (d)
  (let ((td (base_type (type (car d)))))
  (let ((arg_err (arg_check d (list td))))
    (if (equal arg_err nil)
        (if (scalar_typep td)
            (if (Gtruep (Gequal
                          (car d)
                          (std_lower (list (marked 'type_descriptor td)))))
                (marked (lower_pred_error td) (default_value td))
              (marked nil (typed td (sub1 (value (car d))))))
          (marked (pred_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))

(defn std_range (d)
  (let ((arg_err (arg_check d (list (type (car d))))))
    (if (equal arg_err nil)
        (if (mapping_descp (type (car d)))
            ; 2.05 report says base_type for range, but not for domain
            (marked nil (typed (set_desc nil (component_td (type (car d))))
                               (vrange (value (car d)))))
          (marked (range_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_scale (d)
  (let ((arg_err (arg_check d (list (integer_desc) 'type_descriptor))))
    (if (equal arg_err nil)
```

```
        (let ((td (base_type (unmark (cadr d)))))
          (if (scalar_typep td)
              (let ((ok (dtype td (value (car d)))))
                (if (truep ok)
                    (marked nil (typed td (value (car d))))
                (if (errorp ok)
                    (marked ok (default_value td))
                  (marked (scale_int_arg_error (value (car d)) td)
                          (default_value td)))))
            (marked (scale_type_arg_error td)
                    (default_value (integer_desc)))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_size (d)
  (let ((arg_err (arg_check d (list (type (car d))))))
    (if (equal arg_err nil)
        (if (or (mapping_descp (type (car d)))
                (sequence_descp (type (car d)))
                (set_descp (type (car d))))
            (marked nil (typed (integer_desc) (vsize (value (car d)))))
          (marked (size_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_upper (d)
  (let ((arg_err (arg_check d (list 'type_descriptor))))
    (if (equal arg_err nil)
        (let ((td (unmark (car d))))
        (let ((r (tmax td)))
          (if (equal r nil)
              (marked (unbounded_type_error td) (default_value td))
          (if (errorp r)
              (marked r (default_value td))
            (marked nil (typed td r))))))
      (marked arg_err (default_value (integer_desc))))))

(defn std_succ (d)
  (let ((td (base_type (type (car d)))))
  (let ((arg_err (arg_check d (list td))))
    (if (equal arg_err nil)
        (if (scalar_typep td)
            (if (Gtruep (Gequal (car d)
                                (std_upper (list (marked 'type_descriptor
                                                         td)))))
                (marked (upper_succ_error td) (default_value td))
              (marked nil (typed td (add1 (value (car d))))))
          (marked (succ_arg_error) (default_value (integer_desc))))
      (marked arg_err (default_value (integer_desc)))))))



; ***********
;  Variables
; ***********

(defn new_namep (n v)
  (and (not (in_map v n))
       (not (reserved_idp n))))

(defn free_variablep (n v)
  (in_map v n))

(defn free_value (n v)
  (mapped_value v n))

(defn apply_var (fn v d)
  (if (free_variablep fn v)
      (select_op (free_value fn v) d)
    (marked (unknown_name_error fn)
            (default_value (integer_desc)))))
```

```
; ****************
;  Function Calls
; ****************

(defn GF_prec (u)
  ; Don Good tentatively favors replacing result with f (<formals>).
  (subst_tree (mk_entry_value 'result)
              (mk_identifier 'result)
              (prec u)))

(defn eq_opp (op etype)

  (if (or (rule op (prodn (tag 'binary_operator 'op) 'EQ))
          (rule op (prodn (tag 'binary_operator 'op) 'EQUAL)))
      (equality_typep etype)

  (if (rule op (prodn (tag 'binary_operator 'op) 'IFF))
      (boolean_typep etype)

    F)))


(defn f_of_formals (fn fs)
  (if (nlistp fs)
      (mk_expression
        (mk_modified_primary_value (mk_identifier fn)))
    (mk_expression
      (mk_tree 'modified_primary_value
               (list (mk_modified_primary_value (mk_identifier fn))
                     (mk_value_modifiers
                       (namelist_to_actuals (dparam_name_list fs) nil)))))))

(defn function_defn (u ftype)
  (let ((fc (f_of_formals (unit_name u) (formal_dargs u)))
        (result (mk_expression 'result)))
  (let ((e (postc u 'NORMAL)))

    (if (rule e (prodn (tag 'expression 'e)
                       (list (tag 'expression 'e1)
                             (tag 'binary_operator 'op)
                             (tag 'expression 'e2))))
        (if (eq_opp (subtree e 'binary_operator) ftype)
            (if (or (equal (subtree_i e 'expression 1) result)
                    (equal (subtree_i e 'expression 1) fc))
                 (subst_tree fc result (subtree_i e 'expression 2))
            (if (or (equal (subtree_i e 'expression 2) result)
                    (equal (subtree_i e 'expression 2) fc))
                (subst_tree fc result (subtree_i e 'expression 1))
              nil))
          nil)
      nil))))

; Previous definition of function_defn
;(defn function_defn (u ftype)
;  (let ((fc (f_of_formals (unit_name u) (formal_dargs u)))
;       (result (mk_expression 'result)))
;  (let ((e (subst_tree result fc (postc u 'NORMAL))))
;
;    (if (rule e (prodn (tag 'expression 'e)
;                       (list (tag 'expression 'e1)
;                             (tag 'binary_operator 'op)
;                             (tag 'expression 'e2))))
;       (if (eq_opp (subtree e 'binary_operator) ftype)
;           (if (and (equal (subtree_i e 'expression 1) result)
;                    (not (subtreep result (subtree_i e 'expression 2))))
;               (subtree_i e 'expression 2)
;           (if (and (equal (subtree_i e 'expression 2) result)
;                    (not (subtreep result (subtree_i e 'expression 1))))
;               (subtree_i e 'expression 1)
```

```
;             nil))
;         nil)
;      nil))))


(defn has_defn (u ftype)
  (if (equal (kind u) 'function)
      (not (equal (function_defn u ftype) nil))
  (if (equal (kind u) 'constant)
      (not (equal (constant_value_exp u) nil))
    F)))

(defn Gdefn (u ftype)
  ; Need to check has_defn before using this.
  (if (equal (kind u) 'function)
      (function_defn u ftype)
  (if (equal (kind u) 'constant)
      (constant_value_exp u)
    nil)))

(defn formal_type_list (fs sn x)
  (if (nlistp fs)
      nil
    (cons (type_desc (formal_type (car fs)) sn nil x)
          (formal_type_list (cdr fs) sn x))))

(disable reserved_idp)
(disable dparam_name)
(disable param_reserved_error)
(disable dparam_name_list)
(disable duplicate_param_names_error)
(disable access)
(disable function_access_error)

(defn fformals_check (fs)
  (if (nlistp fs)
      nil
  (if (reserved_idp (dparam_name (car fs)))
      (param_reserved_error (car fs))
  (if (or (member (dparam_name (car fs))
                  (dparam_name_list (cdr fs)))
          (equal (dparam_name (car fs)) 'result))
      (duplicate_param_names_error (car fs))
  (if (equal (access (car fs)) 'var)
      (function_access_error (car fs))
    (fformals_check (cdr fs)))))))

(defn farg_check (fs as fsn x)
  (let ((r (fformals_check fs)))
    (if (equal r nil)
        (arg_check as (formal_type_list fs fsn x))
      r)))

(defn bind_args (fs as fsn x)
  (if (nlistp fs)
      (empty_map)
    (add_to_map (bind_args (rcdr fs) (rcdr as) fsn x)
                (dparam_name (rcar fs))
                (marked nil
                        (typed (type_desc (formal_type (rcar fs)) fsn nil x)
                               (value (rcar as)))))))

(defn mk_entry_name (n)
  ; n is a gname, a litatom
  (pack (rcons (unpack n) (ascii_single_quote))))

(defn type_namep (tn sn x)
  (or (member tn '(boolean character integer rational))
      (equal (kind (ref_unit (ref tn sn x))) 'type)))
```

```
(defn type_name_expp (fn ap sn x)
  (and (type_namep fn sn x)
       (nlistp ap)))



; ***********************
;  Expression Evaluation
; ***********************

; Note:  Semantic errors that don't come up during evaluation of expressions
;        are not detected.
;
;        1. Uniqueness of local names requirements are enforced only by the
;           reference resolution mechanism and by requiring unique names for
;           data objects in the environment.  For example, in
;
;               f(x) & all f : t, g(x,f)
;
;           the use of f both as a function name and as a bound identifier is
;           not detected, and in
;
;               function f (i : i) : boolean = pending
;
;           the use of i both as a parameter name and as a type name is not
;           detected.
;
;        2. Errors in conditional exit specifications of functions are
;           detected only if they make it impossible to locate the function
;           definition.  None of the errors in
;
;               function f (x : integer) : boolean =
;               begin
;                 exit case (is normal : (assume result = g(x));
;                            is     c1 : (assume c1(x));
;                            is     c1 : (assume c2(x)));
;               end;
;
;           are detected.
;
;        3. Errors are detected in only the evaluated parts of if expressions.
;           For example, no errors are detected in
;
;               if false then
;                 'a'
;               elif true then
;                 3
;               elif 1/2 then
;                 [set: binary 28]
;               else 4 + "a string"
;               fi
;
;        4. Errors in scope, unit, and name declarations are detected only if
;           they come up in a unit reference.



(defn bound_values (e c x)

  (if (rule e (prodn (tag 'expression 'e)
                     (list 'ALL (tag 'bound_expression 'b))))
      (bound_values (subtree e 'bound_expression) c x)

  (if (rule e (prodn (tag 'expression 'e)
                     (list 'SOME (tag 'bound_expression 'b))))
      (bound_values (subtree e 'bound_expression) c x)

  (if (rule e (prodn (tag 'bound_expression 'b)
                     (list (tag 'identifier_list 'q) 'COLON
```

```
                           (tag 'type_specification 's) 'COMMA
                           (tag 'expression 'e))))
      (marked_typed_value_set
       (type_desc (subtree e 'type_specification) c nil x))

  (if (rule e (prodn (tag 'opt_each_clause 'e)
                     (list 'EACH (tag 'IDENTIFIER 'i) 'COLON
                           (tag 'type_specification 'ts) 'COMMA)))
      (let ((td (type_desc (subtree e 'type_specification) c nil x)))
        (if (bounded_index_typep td)
            (marked_typed_value_set td)
          (each_id_type_error e c)))

    nil))))

  ( (lessp (tree_size e)) ))



; =================================
;  Lemmas for GF Group Termination
; =================================

; Make these lemmas more reasonable if there is time.


; ==============================================
;  Case of Characters Does Not Affect Tree Size
; ==============================================

(prove-lemma upper_case_tree_size (rewrite)
  (implies (ascii_characterp x)
           (equal (tree_size (upper_case x))
                  (tree_size x))))

(prove-lemma uc_list_tree_size (rewrite)
  (implies (ascii_character_listp x)
           (equal (tree_size (uc_list x))
                  (tree_size x)))
  ( (disable upper_case)
    (induct (uc_list x)) ))

(disable upper_case_tree_size)
(disable uc_list_tree_size)


; ==========================================================
;  Else Part of If Expression Is Smaller than If Expression
; ==========================================================

; ------------------------
;  IF is smaller than ELIF
; ------------------------

(prove-lemma lessp_subtrees_imp_lessp_tree_size nil
  (implies (and (treep x) (treep y)
                (lessp (tree_size (subtrees x))
                       (tree_size (subtrees y))))
           (lessp (tree_size x) (tree_size y))))

(prove-lemma same_leaf_imp_same_tree_size1 nil
  (implies (and (leafp x) (leafp y)
                (equal (root x) (root y))
                (equal (lexeme x) (lexeme y)))
           (equal (tree_size x) (tree_size y)))
  ( (disable tokenp *1*tokenp mk_tree root subtrees tree_size)
    (enable leafp lexeme)
    (use (lessp_subtrees_imp_lessp_tree_size)) ))
```

```
(prove-lemma same_leaf_imp_same_tree_size2 nil
  (implies (and (leafp x) (leafp y)
                (equal (root x) (root y))
                (equal (uc_list (lexeme x))
                       (uc_list (lexeme y))))
           (equal (tree_size x) (tree_size y)))
  ( (disable tokenp *1*tokenp)
    (enable leafp lexeme)
    (use (uc_list_tree_size (x (subtrees x)))
         (uc_list_tree_size (x (subtrees y)))) ))

(prove-lemma leaf_equal_imp_tree_size_equal (rewrite)
  (implies (leaf_equal x y)
           (equal (tree_size x) (tree_size y)))
  ( (disable leafp lexeme)
    (use (same_leaf_imp_same_tree_size1)
         (same_leaf_imp_same_tree_size2)) ))

(prove-lemma tree_equal_imp_tree_size_equal nil
  (implies (tree_equal x y)
           (equal (tree_size x) (tree_size y)))
  ( (disable leafp leaf_equal) ))

(prove-lemma elif_leafp (rewrite)
  (implies (and (parse_tree_leafp e)
                (equal (root e) ’elif))
           (tree_equal e (mk_reserved_word ’ELIF))))

(prove-lemma mk_rhs_imp_root (rewrite)
  (implies (and (parse_treep e)
                (equal (mk_rhs e) ’elif))
           (and (parse_tree_leafp e)
                (equal (root e) ’ELIF)))
  ( (disable parse_tree_leafp mk_reserved_word *1*mk_reserved_word tokenp) ))

(prove-lemma if_exp_else_subtrees_car nil
  (implies (rule e (prodn (tag ’if_expression_else_part ’e)
                          (list ’ELIF (tag ’expression ’b) ’THEN
                                (tag ’expression ’p)
                                (tag ’if_expression_else_part ’e2))))
           (tree_equal (car (subtrees e))
                       (mk_reserved_word ’elif)))
  ( (disable untag tokenp parse_tree_leafp
             mk_reserved_word *1*mk_reserved_word)
    (enable rule) ))

(prove-lemma lessp_if_than_elif (rewrite)
  (implies (rule e (prodn (tag ’if_expression_else_part ’e)
                          (list ’ELIF (tag ’expression ’b) ’THEN
                                (tag ’expression ’p)
                                (tag ’if_expression_else_part ’e2))))
           (lessp (tree_size (mk_reserved_word ’if))
                  (tree_size (car (subtrees e)))))
  ( (disable mk_reserved_word rule tree_equal tree_size)
    (use (tree_equal_imp_tree_size_equal (x (car (subtrees e)))
                                         (y (mk_reserved_word ’elif)))
         (if_exp_else_subtrees_car)) ))


; --------------------
;  Subtrees Are a List
; --------------------

(prove-lemma listp_if_exp_else_subtrees (rewrite)
  (implies (rule e (prodn (tag ’if_expression_else_part ’e)
                          (list ’ELIF (tag ’expression ’b) ’THEN
                                (tag ’expression ’p)
                                (tag ’if_expression_else_part ’e2))))
           (listp (subtrees e)))
```

```
  ( (disable untag tokenp parse_tree_leafp)
    (enable rule) ))


; -------------------------
;  The Else Part Is Smaller
; -------------------------

(prove-lemma lessp_car_imp_lessp_tree_size (rewrite)
  (implies (and (listp x) (listp y)
                (lessp (tree_size (car x)) (tree_size (car y)))
                (equal (tree_size (cdr x)) (tree_size (cdr y))))
           (lessp (tree_size x) (tree_size y))))

(prove-lemma lessp_mk_tree_car_imp_lessp_tree_size (rewrite)
  (implies (and (treep e) (listp (subtrees e))
                (lessp (tree_size k)
                       (tree_size (car (subtrees e)))))
           (lessp (tree_size (mk_tree nt (cons k (cdr (subtrees e)))))
                  (tree_size e)))
  ( (disable tree_size)
    (use (lessp_subtrees_imp_lessp_tree_size
           (x (mk_tree nt (cons k (cdr (subtrees e)))))
           (y e))) ))

(disable lessp_car_imp_lessp_tree_size)

(prove-lemma leq_if_exp_else_part nil
  (implies (not (lessp (tree_size (if_else_exp e))
                       (tree_size e)))
           (equal (tree_size (if_else_exp e))
                  (tree_size e)))
  ( (disable tree_size mk_reserved_word *1*mk_reserved_word)
    (induct (if_else_exp e)) ))

(prove-lemma lessp_if_exp_else_part (rewrite)
  (implies (rule e (prodn x y))
           (lessp (tree_size (if_else_exp (subtree e n)))
                  (tree_size e)))
  ( (disable tree_size if_else_exp)
    (use (leq_if_exp_else_part (e (subtree e n)))
         (rule_imp_lessp_subtree_size (u e) (x x) (y y) (z n))) ))


; =======================================
;  Gname Is Smaller Than Identifier Tree
; =======================================

(prove-lemma identifierp_imp_treep nil
  (implies (identifierp e)
           (treep e))
  ( (disable identifier_lexemep lexeme root tokenp *1*tokenp subtrees
             ascii_character_listp)
    (enable identifierp) ))

(prove-lemma lessp_gname_tree_size_0 nil
  (implies (treep e)
           (lessp (tree_size (gname e))
                  (tree_size e)))
  ( (disable identifierp uc_list lexeme) ))

(prove-lemma lessp_gname_tree_size (rewrite)
  (implies (identifierp e)
           (lessp (tree_size (gname e))
                  (tree_size e)))
  ( (disable identifierp tree_size gname)
    (use (identifierp_imp_treep)
         (lessp_gname_tree_size_0)) ))
```

```
; ===================================================
;  Object Name Is Smaller Than Containing Expression
; ===================================================

(prove-lemma leq_object_name_tree_size (rewrite)
  (implies (not (lessp (tree_size (object_name e))
                       (tree_size e)))
           (equal (tree_size (object_name e))
                  (tree_size e)))
  ( (disable tree_size identifierp gname)
    (induct (object_name e)) ))

(disable leq_object_name_tree_size)

(prove-lemma lessp_object_name_tree_size (rewrite)
  (implies (rule e (prodn x y))
           (lessp (tree_size (object_name (subtree e n)))
                  (tree_size e)))
  ( (disable tree_size object_name)
    (use (leq_object_name_tree_size (e (subtree e n)))
         (rule_imp_lessp_subtree_size (u e) (x x) (y y) (z n))) ))


; ==================================================
;  Argument List Smaller Than Containing Expression
; ==================================================

(prove-lemma leq_arg_list_tree_size (rewrite)
  (implies (not (lessp (tree_size (arg_list e))
                       (tree_size e)))
           (equal (tree_size (arg_list e))
                  (tree_size e)))
  ( (disable tree_size)
    (induct (arg_list e)) ))

(disable leq_arg_list_tree_size)

(prove-lemma lessp_arg_list_tree_size (rewrite)
  (implies (rule e (prodn x y))
           (lessp (tree_size (arg_list (subtree e n)))
                  (tree_size e)))
  ( (disable tree_size arg_list)
    (use (leq_arg_list_tree_size (e (subtree e n)))
         (rule_imp_lessp_subtree_size (u e) (x x) (y y) (z n))) ))


; =========================================================
;  Bound Identifier Is Smaller Than Containing Expression
; =========================================================

(prove-lemma leq_bound_id_tree_size (rewrite)
  (implies (not (lessp (tree_size (bound_id e))
                       (tree_size e)))
           (equal (tree_size (bound_id e))
                  (tree_size e)))
  ( (disable tree_size identifierp gname)
    (induct (bound_id e)) ))

(disable leq_bound_id_tree_size)

(prove-lemma lessp_bound_id_tree_size (rewrite)
  (implies (rule e (prodn x y))
           (lessp (tree_size (bound_id (subtree e n)))
                  (tree_size e)))
  ( (disable tree_size bound_id)
    (use (leq_bound_id_tree_size (e (subtree e n)))
         (rule_imp_lessp_subtree_size (u e) (x x) (y y) (z n))) ))
```

```
(disable *1*Gfalse)
(disable *1*Gizero)
(disable *1*Gtrue)
(disable *1*boolean_desc)
(disable *1*integer_desc)
(disable *1*mk_unary_operator)
(disable GF_prec)
(disable Gand)
(disable Gchar)
(disable Gdefn)
(disable Gfalse)
(disable Gizero)
(disable Gle)
(disable Gmap_insert)
(disable Gmapomit)
(disable Gor)
(disable Grange_elements)
(disable Gseq_insert_before)
(disable Gseq_insert_behind)
(disable Gseqomit)
(disable Gset_or_seq)
(disable Gstring_seq)
(disable Gtrue)
(disable Gtruep)
(disable arg_list)
(disable add_to_map)
(disable apply_binary_op)
(disable apply_unary_op)
(disable apply_var)
(disable bad_value_modifiers_error)
(disable base_type)
(disable bind_args)
(disable boolean_desc)
(disable bound_id)
(disable bound_values)
(disable cdr_quantified_exp)
(disable character_valuep)
(disable condition_params_error)
(disable default_value)
(disable determinate)
(disable digit_listp)
(disable each_clausep)
(disable entry_name)
(disable entry_not_true_error)
(disable entry_valuep)
(disable errorp)
(disable farg_check)
(disable fn_call_formp)
(disable formal_dargs)
(disable free_variablep)
(disable gname)
(disable has_defn)
(disable identifierp)
(disable if_else_exp)
(disable in_type)
(disable indeterminate)
(disable indeterminate_fn_result_error)
(disable integer_desc)
(disable kind)
(disable length)
(disable minteger)
(disable mk_entry_name)
(disable mk_identifier)
(disable mk_unary_operator)
(disable n_too_small)
(disable name_already_in_use_error)
(disable new_namep)
(disable no_function_defn_error)
(disable not_expression_error)
```

```
(disable not_function_or_const_error)
(disable object_name)
(disable put_op)
(disable rcar)
(disable rcdr)
(disable rcons)
(disable record_get)
(disable ref)
(disable result_type)
(disable select_op)
(disable std_domain)
(disable std_first)
(disable std_initial)
(disable std_last)
(disable std_lower)
(disable std_max)
(disable std_min)
(disable std_nonfirst)
(disable std_nonlast)
(disable std_null)
(disable std_ord)
(disable std_pred)
(disable std_range)
(disable std_scale)
(disable std_size)
(disable std_succ)
(disable std_upper)
(disable string_valuep)
(disable subsequence_get)
(disable tree_size)
(disable type)
(disable type_desc)
(disable type_name_expp)
(disable typed)
(disable unmark)
(disable value)


(do-mutual '(


; ***************************
;  Set/Sequence Constructors
; ***************************

(defn GF_element_list (e c v n x)

  (if (rule e (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GF_element_list (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'value_list 'v)))
      (GF_element_list (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'range_limits 'r)))
      (GF_element_list (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (Grange_elements (GF (subtree_i e 'expression 1) c v n x)
                       (GF (subtree_i e 'expression 2) c v n x))

  (if (rule e (prodn (tag 'value_list 'v)
                     (tag 'expression 'e)))
      (rcons nil (GF (subtree e 'expression) c v n x))
```

```
  (if (rule e (prodn (tag 'value_list 'v)
                     (list (tag 'value_list 'v2) 'COMMA
                           (tag 'expression 'e))))
      (rcons (GF_element_list (subtree e 'value_list) c v n x)
             (GF (subtree e 'expression) c v n x))

    nil))))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size e)))
                     (count c))) )) ; (count c) is a place filler

(defn GF_element_type (e c v n x)

  (if (rule e (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GF_element_type (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'value_list 'v)))
      (GF_element_type (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'range_limits 'r)))
      (GF_element_type (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (base_type (type (GF (subtree_i e 'expression 1) c v n x)))

  (if (rule e (prodn (tag 'value_list 'v)
                     (tag 'expression 'e)))
      (base_type (type (GF (subtree e 'expression) c v n x)))

  (if (rule e (prodn (tag 'value_list 'v)
                     (list (tag 'value_list 'v2) 'COMMA
                           (tag 'expression 'e))))
      (GF_element_type (subtree e 'value_list) c v n x)

    nil))))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size e)))
                     (count c))) )) ; (count c) is a place filler



; ************************
;  Quantified expressions
; ************************

(defn GF_all (id vs e c v n x)
  (if (nlistp vs)
      (Gtrue)
  (if (new_namep id v)
      (if (zerop n)
          (marked (n_too_small)
                  (default_value (boolean_desc)))
        (Gand (GF_all id (rcdr vs) e c v n x)
              (GF e c (add_to_map v id (rcar vs)) (sub1 n) x)))
    (marked (name_already_in_use_error id)
            (default_value (boolean_desc)))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size id)))
                     (count vs))) ))

(defn GF_some (id vs e c v n x)
  (if (nlistp vs)
      (Gfalse)
    (if (new_namep id v)
```

```
      (if (zerop n)
          (marked (n_too_small)
                  (default_value (boolean_desc)))
        (Gor (GF_some id (rcdr vs) e c v n x)
             (GF e c (add_to_map v id (rcar vs)) (sub1 n) x)))
      (marked (name_already_in_use_error id)
              (default_value (boolean_desc)))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size id)))
                     (count vs))) ))


; ********************
;  Value Modifications
; ********************

(defn GF_each (id vs bv e c v n x)
  ; e is the <component modification>
  (if (nlistp vs)
      bv
  (if (new_namep id v)
      (if (zerop n)
          (marked (n_too_small)
                  (default_value (base_type (type bv))))
        (GF_each id (cdr vs)
                 (GF_modifiers bv e c (add_to_map v id (car vs)) (sub1 n) x)
                 e c v n x))
    (marked (name_already_in_use_error id)
            (default_value (base_type (type bv))))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size e)))
                     (count vs))) ))

(defn GF_adp (e c v n x)

  (if (rule e (prodn (tag 'arg_list 'as)
                     (list 'OPEN_PAREN (tag 'value_list 'vs)
                           'CLOSE_PAREN)))
      (GF_adp (subtree e 'value_list) c v n x)

  (if (rule e (prodn (tag 'value_list 'vs)
                     (tag 'expression 'e)))
      (rcons nil (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'value_list 'vs)
                     (list (tag 'value_list 'vs2)
                           'COMMA (tag 'expression 'e))))
      (rcons (GF_adp (subtree e 'value_list) c v n x)
             (GF (subtree e 'expression) c v n x))
    nil)))

  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size e)))
                     (count c))) )) ; (count c) is a place filler

(defn GF_selectors (e c v n x)

  (if (rule e (prodn (tag 'selector_list 's)
                     (tag 'component_selectors 's2)))
      (GF_selectors (subtree e 'component_selectors) c v n x)

  (if (rule e (prodn (tag 'selector_list 's)
                     (list (tag 'selector_list 's2)
                           (tag 'component_selectors 's3))))
      (append (GF_selectors (subtree e 'selector_list) c v n x)
              (GF_selectors (subtree e 'component_selectors) c v n x))

  (if (rule e (prodn (tag 'component_selectors 's)
                     (list 'DOT (tag 'IDENTIFIER 'fn))))
```

```
      (list (marked 'field_name
                    (gname (subtree e 'IDENTIFIER))))

  (if (rule e (prodn (tag 'component_selectors 's)
                     (tag 'arg_list 'd)))
      (GF_adp (subtree e 'arg_list) c v n x)

  (if (rule e (prodn (tag 'arg_list 'as)
                     (list 'OPEN_PAREN (tag 'value_list 'vs)
                           'CLOSE_PAREN)))
      (GF_adp (subtree e 'value_list) c v n x)

    nil)))))

  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size e)))
                     (count c))) )) ; (count c) is a place filler

(defn GF_modifiers (bv e c v n x)
  ; e is the <value modifiers>

  (if (rule e (prodn (tag 'value_modifiers 'm)
                     (tag 'component_selectors 's)))
      (GF_modifiers bv (subtree e 'component_selectors) c v n x)

  (if (rule e (prodn (tag 'component_selectors 's)
                     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (record_get bv
                  (marked 'field_name
                          (gname (subtree e 'IDENTIFIER)))
                  (type bv))

  (if (rule e (prodn (tag 'component_selectors 's)
                     (tag 'arg_list 'd)))
      (select_op bv (GF_adp (subtree e 'arg_list) c v n x))

  (if (rule e (prodn (tag 'value_modifiers 'm)
                     (tag 'range 'r)))
      (GF_modifiers bv (subtree e 'range) c v n x)

  (if (rule e (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2)
                           'CLOSE_PAREN)))
      (GF_modifiers bv (subtree e 'range_limits) c v n x)

  (if (rule e (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (subsequence_get bv
                       (GF (subtree_i e 'expression 1) c v n x)
                       (GF (subtree_i e 'expression 2) c v n x))

  (if (rule e (prodn (tag 'value_modifiers 'm)
                     (tag 'value_alterations 'a)))
      (GF_modifiers bv (subtree e 'value_alterations) c v n x)

  (if (rule e (prodn (tag 'value_alterations 'a)
                     (list 'WITH 'OPEN_PAREN
                           (tag 'component_alterations_list 'al)
                           'CLOSE_PAREN)))
      (GF_modifiers bv (subtree e 'component_alterations_list) c v n x)

  (if (rule e (prodn (tag 'component_alterations_list 'al)
                     (tag 'component_alterations 'a)))
      (GF_modifiers bv (subtree e 'component_alterations) c v n x)

  (if (rule e (prodn (tag 'component_alterations_list 'al)
                     (list (tag 'component_alterations_list 'al2)
                           'SEMI_COLON (tag 'component_alterations 'a))))
```

```
      (GF_modifiers (GF_modifiers bv (subtree e 'component_alterations_list)
                                  c v n x)
                    (subtree e 'component_alterations) c v n x)

  (if (rule e (prodn (tag 'component_alterations 'as)
                     (list (tag 'opt_each_clause 'e)
                           (tag 'component_assignment 'a))))
      (if (each_clausep (subtree e 'opt_each_clause))
          (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
            (if (errorp vs)
                (marked vs (default_value (base_type (type bv))))
              (GF_each (bound_id (subtree e 'opt_each_clause))
                       vs bv (subtree e 'component_assignment) c v n x)))
        (GF_modifiers bv (subtree e 'component_assignment) c v n x))

  (if (rule e (prodn (tag 'component_alterations 'as)
                     (list (tag 'opt_each_clause 'e)
                           (tag 'component_creation 'c))))
      (if (each_clausep (subtree e 'opt_each_clause))
          (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
            (if (errorp vs)
                (marked vs (default_value (base_type (type bv))))
              (GF_each (bound_id (subtree e 'opt_each_clause))
                       vs bv (subtree e 'component_creation) c v n x)))
        (GF_modifiers bv (subtree e 'component_creation) c v n x))

  (if (rule e (prodn (tag 'component_alterations 'as)
                     (list (tag 'opt_each_clause 'e)
                           (tag 'component_deletion 'd))))
      (if (each_clausep (subtree e 'opt_each_clause))
          (let ((vs (bound_values (subtree e 'opt_each_clause) c x)))
            (if (errorp vs)
                (marked vs (default_value (base_type (type bv))))
              (GF_each (bound_id (subtree e 'opt_each_clause))
                       vs bv (subtree e 'component_deletion) c v n x)))
        (GF_modifiers bv (subtree e 'component_deletion) c v n x))

  (if (rule e (prodn (tag 'component_assignment 'a)
                     (list (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (put_op bv
              (GF_selectors (subtree e 'selector_list) c v n x)
              (GF (subtree e 'expression) c v n x))

  (if (rule e (prodn (tag 'component_creation 'c)
                     (list 'BEFORE (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
            (u (GF (subtree e 'expression) c v n x)))
        (put_op bv (rcdr s)
                (Gseq_insert_before (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_creation 'c)
                     (list 'BEHIND (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
            (u (GF (subtree e 'expression) c v n x)))
        (put_op bv (rcdr s)
                (Gseq_insert_behind (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_creation 'c)
                     (list 'INTO (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x))
            (u (GF (subtree e 'expression) c v n x)))
        (put_op bv (rcdr s)
                (Gmap_insert (select_op bv (rcdr s)) (rcar s) u)))

  (if (rule e (prodn (tag 'component_deletion 'd)
```

```
                     (list 'SEQOMIT (tag 'selector_list 's))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x)))
        (put_op bv (rcdr s)
                (Gseqomit (select_op bv (rcdr s)) (rcar s))))

  (if (rule e (prodn (tag 'component_deletion 'd)
                     (list 'MAPOMIT (tag 'selector_list 's))))
      (let ((s (GF_selectors (subtree e 'selector_list) c v n x)))
        (put_op bv (rcdr s)
                (Gmapomit (select_op bv (rcdr s)) (rcar s))))

    (marked (bad_value_modifiers_error e)
            (default_value (base_type (type bv))))))))))))))))))))))

  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size e)))
                     (count c))) )) ; (count c) is a place filler



; ************************************
;  Name references and function calls
; ************************************

(defn apply_fun (fn d sn n x)
  (let ((h (car (ref fn sn x)))  ; scope fn is declared in
        (u (cdr (ref fn sn x)))) ; the function declaration
    (if (or (equal (kind u) 'function)
            (equal (kind u) 'constant))
        (let ((ftype (type_desc (result_type u) h nil x))
              (fs (formal_dargs u)))                  ; formals
        (let ((a (if (equal (length fs) 0) nil d))    ; actuals
              (s (if (equal (length fs) 0) d nil)))   ; selectors
          (if (zerop (fix n))
              (marked (n_too_small)
                      (default_value ftype))
            (select_op
             (let ((arg_err (farg_check fs a h x)))
               (if (equal arg_err nil)
                   (let ((v (add_to_map (bind_args fs a h x)
                                        (mk_entry_name 'result)
                                        (std_initial
                                         (list (marked 'type_descriptor
                                                       ftype))))))
                     (if (Gtruep (GF (GF_prec u) h v (sub1 n) x))
                         (if (has_defn u ftype)
                             (let ((r (GF (Gdefn u ftype)
                                          h v (sub1 n) x)))
                               (if (and (determinate r)
                                        (truep (in_type ftype r)))
                                   (marked nil
                                           (typed ftype (value r)))
                                 (marked
                                   (indeterminate_fn_result_error fn sn)
                                   (default_value ftype))))
                           (marked (no_function_defn_error fn sn)
                                   (default_value ftype)))
                       (marked (entry_not_true_error fn sn)
                               (default_value ftype))))
                 (marked arg_err (default_value ftype))))
             s))))
    (if (equal (kind u) 'error)
        (marked u (default_value (integer_desc)))
      (marked (not_function_or_const_error fn sn)
              (default_value (integer_desc))))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size fn)))
                     (tree_size fn))) ))
```

```
(defn Gapply (fn ap sn v n x)
  (if (free_variablep fn v)
      (apply_var fn v ap)
  (if (equal fn 'false)
      (select_op (Gfalse) ap)
  (if (equal fn 'true)
      (select_op (Gtrue) ap)
  (if (type_name_expp fn ap sn x)
      (marked 'type_descriptor
              (type_desc (mk_identifier fn) sn nil x))
  (if (equal fn 'domain)
      (std_domain ap)
  (if (equal fn 'first)
      (std_first ap)
  (if (equal fn 'initial)
      (std_initial ap)
  (if (equal fn 'last)
      (std_last ap)
  (if (equal fn 'lower)
      (std_lower ap)
  (if (equal fn 'max)
      (std_max ap)
  (if (equal fn 'min)
      (std_min ap)
  (if (equal fn 'nonfirst)
      (std_nonfirst ap)
  (if (equal fn 'nonlast)
      (std_nonlast ap)
  (if (equal fn 'null)
      (std_null ap)
  (if (equal fn 'ord)
      (std_ord ap)
  (if (equal fn 'pred)
      (std_pred ap)
  (if (equal fn 'range)
      (std_range ap)
  (if (equal fn 'scale)
      (std_scale ap)
  (if (equal fn 'size)
      (std_size ap)
  (if (equal fn 'succ)
      (std_succ ap)
  (if (equal fn 'upper)
      (std_upper ap)
    (apply_fun fn ap sn n x)))))))))))))))))))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size fn)))
                     (add1 (tree_size fn)))) ))


(defn GF (e c v n x)
  ; The meta-function GF(e,c,v,n,x) gives the value that results when the
  ; expression e is evaluated, in the context of scope c with free variables
  ; bound as determined in the environment v, by at most n applications of
  ; functions described by the Gypsy sentence x.
  ;
  ; The domain and range of GF(e,c,v,n,x) are as follows:
  ;
  ;   e is the parse tree representing the expression to be evaluated.
  ;   c is the (litatom) name of the Gypsy scope in which e is evaluated.
  ;   v is the name-value mapping that maps names of free variables
  ;     into their (marked typed) values.
  ;   n is the maximum allowed depth of Gypsy function calls and quantifiers
  ;   x is the parse tree representing the Gypsy sentence that is being
  ;     interpreted.  This is the complete sentence containing the list of all
  ;     available scopes.
  ;   GF(e,c,v,n,x) is the marked, typed value that results from evaluating e.
  ;
```

```
; ************************
;
;    <expression> ::= ...
;
; ************************

(if (rule e (prodn (tag 'expression 'e)
                   (tag 'modified_primary_value 'm)))
    (GF (subtree e 'modified_primary_value) c v n x)

(if (rule e (prodn (tag 'expression 'e)
                   (list 'ALL (tag 'bound_expression 'b))))
    (let ((vs (bound_values e c x)))
      (if (errorp vs)
          (marked vs (default_value (boolean_desc)))
        (GF_all (bound_id (subtree e 'bound_expression))
                vs (cdr_quantified_exp e) c v n x)))

(if (rule e (prodn (tag 'expression 'e)
                   (list 'SOME (tag 'bound_expression 'b))))
    (let ((vs (bound_values e c x)))
      (if (errorp vs)
          (marked vs (default_value (boolean_desc)))
        (GF_some (bound_id (subtree e 'bound_expression))
                 vs (cdr_quantified_exp e) c v n x)))

(if (rule e (prodn (tag 'expression 'e)
                   (list (tag 'unary_operator 'op) (tag 'expression 'e2))))
    (apply_unary_op (subtree e 'unary_operator)
                    (GF (subtree e 'expression) c v n x))

(if (rule e (prodn (tag 'expression 'e)
                   (list (tag 'expression 'e1) (tag 'binary_operator 'op)
                         (tag 'expression 'e2))))
    (apply_binary_op (subtree e 'binary_operator)
                     (GF (subtree_i e 'expression 1) c v n x)
                     (GF (subtree_i e 'expression 2) c v n x))


; ********************************
;
;    <modified primary value> ::=
;
; ********************************

(if (rule e (prodn (tag 'modified_primary_value 'm)
                   (tag 'primary_value 'p)))
    (GF (subtree e 'primary_value) c v n x)

(if (rule e (prodn (tag 'modified_primary_value 'm)
                   (list (tag 'modified_primary_value 'm2)
                         (tag 'value_modifiers 'vm))))
    (if (fn_call_formp e)
        (Gapply (object_name (subtree e 'modified_primary_value))
                (GF_adp (arg_list (subtree e 'value_modifiers))
                        c v n x)
                c v n x)
      (GF_modifiers (GF (subtree e 'modified_primary_value) c v n x)
                    (subtree e 'value_modifiers) c v n x))

(if (rule e (prodn (tag 'modified_primary_value 'm)
                   (list (tag 'modified_primary_value 'm2)
                         (tag 'actual_condition_parameters 'cp))))
    ; Condition parameters allowed only in <specification expression>
    ; with validation directive (p. 35, Gypsy 2.05 Report).
    ; **** Weed them out before calling GF in that case. ****
    (let ((r (GF (subtree e 'modified_primary_value) c v n x)))
      (marked (condition_params_error e)
              (default_value (type r))))
```

```
; **********************
;
;    <primary value> ::=
;
; **********************

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'literal_value 'l)))
    (GF (subtree e 'literal_value) c v n x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'set_or_sequence_value 's)))
    (GF (subtree e 'set_or_sequence_value) c v n x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'ENTRY_VALUE 'e)))
    (GF (subtree e 'ENTRY_VALUE) c v n x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'IDENTIFIER 'on)))
    (GF (subtree e 'IDENTIFIER) c v n x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'if_expression 'i)))
    (GF (subtree e 'if_expression) c v n x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (list 'OPEN_PAREN (tag 'expression 'e) 'CLOSE_PAREN)))
    (GF (subtree e 'expression) c v n x)


; ---------------------------------------------------------------------------
;
;  From here down to parse tree leaves, clauses are in alphabetical order
;  by the left-hand side of the productions.  Everything that is a parse
;  tree for an expression should be covered.
;
; ---------------------------------------------------------------------------


; **********************
;
;    <constant body> ::=
;
; **********************

(if (rule e (prodn (tag 'constant_body 'b)
                   (tag 'expression 'e)))
    (GF (subtree e 'expression) c v n x)


; **********************
;
;    <if expression> ::=
;
; **********************

(if (rule e (prodn (tag 'if_expression 'i)
                   (list 'IF (tag 'expression 'b) 'THEN
                         (tag 'expression 'p)
                         (tag 'if_expression_else_part 'e))))

    ; Note:  this does not require all potential value expressions to be the
    ; same type or all boolean expressions to be boolean.
    (let ((bv (GF (subtree_i e 'expression 1) c v n x)))
      (if (indeterminate bv)
          (marked (mark bv)
                  (default_value
                    (type (GF (subtree_i e 'expression 2)
```

```
                              c v n x))))
       (if (truep (in_type (boolean_desc) bv))
           (if (Gtruep bv)
               (GF (subtree_i e 'expression 2) c v n x)
             (GF (if_else_exp (subtree e 'if_expression_else_part))
                 c v n x))
         (marked (if_test_not_boolean_error e c)
                 (default_value
                   (type (GF (subtree_i e 'expression 2)
                             c v n x)))))))


 ; ***********************
 ;
 ;    <literal value> ::=
 ;
 ; ***********************

 (if (rule e (prodn (tag 'literal_value 'l)
                    (tag 'CHARACTER_VALUE 'ch)))
     (GF (subtree e 'CHARACTER_VALUE) c v n x)

 (if (rule e (prodn (tag 'literal_value 'l)
                    (tag 'number 'n)))
     (GF (subtree e 'number) c v n x)

 (if (rule e (prodn (tag 'literal_value 'l)
                    (tag 'STRING_VALUE 's)))
     (GF (subtree e 'STRING_VALUE) c v n x)


 ; ****************
 ;
 ;    <number> ::=
 ;
 ; ****************

 (if (rule e (prodn (tag 'number 'n)
                    (tag 'DIGIT_LIST 's)))
     (minteger e)

 (if (rule e (prodn (tag 'number 'n)
                    (list (tag 'base 'b) (tag 'DIGIT_LIST 's))))
     (minteger e)


 ; *****************************************
 ;
 ;    <pre-computable label expression> ::=
 ;
 ; *****************************************

 (if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                    (tag 'number 'n)))
     (GF (subtree e 'number) c v n x)

 (if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                    (list 'MINUS (tag 'number 'n))))
     (apply_unary_op (mk_unary_operator 'MINUS)
                     (GF (subtree e 'number) c v n x))

 (if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                    (tag 'CHARACTER_VALUE 'ch)))
     (GF (subtree e 'CHARACTER_VALUE) c v n x)

 (if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                    (tag 'IDENTIFIER 'i)))
     (GF (subtree e 'IDENTIFIER) c v n x)
```

```
  ; ********************************
  ;
  ;    <set or sequence value> ::=
  ;
  ; ********************************

  (if (rule e (prodn (tag 'set_or_sequence_value 's)
                     (list 'OPEN_PAREN (tag 'set_or_seq_mark 'm)
                           (tag 'element_list 'e) 'CLOSE_PAREN)))
      (Gset_or_seq (subtree e 'set_or_seq_mark)
                   (GF_element_list (subtree e 'element_list) c v n x)
                   (GF_element_type (subtree e 'element_list) c v n x))

  (if (rule e (prodn (tag 'set_or_sequence_value 's)
                     (tag 'range 'r)))
      (Gset_or_seq nil
                   (GF_element_list (subtree e 'range) c v n x)
                   (GF_element_type (subtree e 'range) c v n x))


  ; *********************
  ;
  ;    PARSE TREE LEAVES
  ;
  ; *********************

  (if (character_valuep e)
      (Gchar e)

  (if (digit_listp e)
      (minteger e)

  (if (entry_valuep e)
      (apply_var (entry_name e) v nil)

  (if (identifierp e)
      (Gapply (gname e) nil c v n x)

  (if (string_valuep e)
      (Gstring_seq e)

    (marked (not_expression_error e) (default_value (integer_desc)))

    ))))))))))))))))))))))))))))))
  ( (ord-lessp (cons (cons (add1 n)
                           (add1 (tree_size e)))
                     (count c))) )) ; (count c) is a place filler

))



; *********************
;  The Meta Function F
; *********************

(defn meta_F (e c v n x)
  (let ((pTe (pT e 'expression))
        (pTx (pT x 'program_description)))
    (if (equal pTe nil)
        (marked 'Expression_Syntax_Error nil)
    (if (equal pTx nil)
        (marked 'Program_Description_Syntax_Error nil)
      (GF pTe c v n pTx)))))
```

D.2 Functions Defining P

The function **P** referred to in the chapters of this report is the function **meta_P**.

```
; ********
;  Errors
; ********

(defn entry_not_boolean_error (e sn)
  (mk_error (list 'The 'entry 'specification e 'in 'scope sn 'is 'not 'a
                 'boolean-valued 'expression)))

(defn exit_label_error (e sn)
  (mk_error (list 'The 'exit 'specification e 'in 'scope sn 'has 'duplicate
                 'or 'unknown 'exit 'labels)))

(defn exit_not_boolean_error (e sn)
  (mk_error (list 'The 'exit 'specification e 'in 'scope sn 'is 'not 'a
                 'boolean-valued 'expression)))



; *************************
;  Parse Tree Constructors
; *************************

(defn mk_opt_condition_handlers (x)
  (mk_tree 'opt_condition_handlers (if (equal x nil) (mk_empty) x)))

(defn mk_signal_stmt (c)
  (mk_tree 'signal_statement
         (list (mk_reserved_word 'SIGNAL) (mk_identifier c))))

(defn mk_elif_into_if_statement (x)
  ;; This expects a tree rooted at 'if_composition_else_part which
  ;; matches rule
  ;; <if composition else part> ::=  ELIF <expression> THEN
  ;;                                      <opt internal statements>
  ;;                                      <if composition else part>
  (mk_tree 'if_composition
         (list (mk_reserved_word 'IF)
               (subtree x 'expression)
               (mk_reserved_word 'THEN)
               (subtree x 'opt_internal_statements)
               (subtree x 'if_composition_else_part)
               (mk_opt_condition_handlers nil)
               (mk_reserved_word 'END))))

(defn component_selectors (m)
  (if (rule m (prodn (tag 'value_modifiers 'm)
                     (tag 'component_selectors 's)))
      (subtree m 'component_selectors)
    nil))

(defn extend_name_selectors (ne cs)

  (if (not (equal (root cs) 'component_selectors))
      nil

  (if (rule ne (prodn (tag 'name_expression 'e)
                      (tag 'IDENTIFIER 'i)))
      (mk_tree 'name_expression
             (list (subtree ne 'IDENTIFIER)
                   (mk_tree 'selector_list cs)))

  (if (rule ne (prodn (tag 'name_expression 'e)
                      (list (tag 'IDENTIFIER 'i)
                            (tag 'selector_list 'ss))))
```

```
      (mk_tree 'name_expression
               (list (subtree ne 'IDENTIFIER)
                     (mk_tree 'selector_list
                              (list (subtree ne 'selector_list) cs))))

    nil))))

(defn mk_name_expression (e)

  (if (equal (root e) 'name_expression)
      e

  (if (rule e (prodn (tag 'expression 'e)
                     (tag 'modified_primary_value 'm)))
      (mk_name_expression (subtree e 'modified_primary_value))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
                     (tag 'primary_value 'p)))
      (mk_name_expression (subtree e 'primary_value))

  (if (rule e (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'value_modifiers 'vm))))
      (extend_name_selectors
        (mk_name_expression (subtree e 'modified_primary_value))
        (component_selectors (subtree e 'value_modifiers)))

  (if (rule e (prodn (tag 'primary_value 'p)
                     (tag 'IDENTIFIER 'on)))
      (mk_name_expression (subtree e 'IDENTIFIER))

  (if (identifierp e)
      (mk_tree 'name_expression e)

    nil))))))

  ( (lessp (tree_size e)) ))



; ***********************************************
;  Parse Tree Extraction and Recognizer Functions
; ***********************************************

(defn case_labels (m)

  (if (rule m (prodn (tag 'case_composition 's)
                     (list 'CASE (tag 'expression 'e)
                           (tag 'case_composition_body 'b)
                           (tag 'opt_condition_handlers 'c) 'END)))
      (case_labels (subtree m 'case_composition_body))

  (if (rule m (prodn (tag 'case_composition_body 'b)
                     'empty))
      nil

  (if (rule m (prodn (tag 'case_composition_body 'b)
                     (list 'ELSE 'COLON (tag 'opt_internal_statements 'ss))))
      nil

  (if (rule m (prodn (tag 'case_composition_body 'b)
                     (list 'IS (tag 'case_labels 'cs) 'COLON
                           (tag 'opt_internal_statements 'ss)
                           (tag 'case_composition_body 'b2))))
      (append (case_labels (subtree m 'case_labels))
              (case_labels (subtree m 'case_composition_body)))

  (if (rule m (prodn (tag 'case_labels 'cs)
                     (tag 'pre_computable_label_expression 'e)))
```

```
      (rcons nil (subtree m 'pre_computable_label_expression))

  (if (rule m (prodn (tag 'case_labels 'cs)
                     (list (tag 'case_labels 'cs2) 'COMMA
                           (tag 'pre_computable_label_expression 'e))))
      (rcons (case_labels (subtree m 'case_labels))
             (subtree m 'pre_computable_label_expression))

    nil))))))

  ( (lessp (tree_size m)) ))


(defn constant_body (u)

  (if (rule u (prodn (tag 'constant_declaration 'd)
                     (list 'CONST (tag 'IDENTIFIER 'cn)
                           'COLON (tag 'type_specification 'rt)
                           'COLON_EQUAL (tag 'constant_body 'b))))
      (subtree u 'constant_body)

    nil))

(defn id_list (d)

  (if (rule d (prodn (tag 'identifier_list 'is)
                     (list (tag 'identifier_list 'is2) 'COMMA
                           (tag 'IDENTIFIER 'i))))
      (rcons (id_list (subtree d 'identifier_list))
             (id_list (subtree d 'IDENTIFIER)))

  (if (rule d (prodn (tag 'identifier_list 'is)
                     (tag 'IDENTIFIER 'i)))
      (rcons nil (id_list (subtree d 'IDENTIFIER)))

  (if (identifierp d)
      (gname d)

    nil)))

  ( (lessp (tree_size d)) ))

(defn actual_cargs (m)

  (if (rule m (prodn (tag 'procedure_statement 's)
                     (list (tag 'IDENTIFIER 'pn)
                           (tag 'arg_list 'dp)
                           (tag 'opt_actual_condition_parameters 'cp))))
      (actual_cargs (subtree m 'opt_actual_condition_parameters))

  (if (rule m (prodn (tag 'modified_primary_value 'm)
                     (list (tag 'modified_primary_value 'm2)
                           (tag 'actual_condition_parameters 'cp))))
      (actual_cargs (subtree m 'actual_condition_parameters))

  (if (rule m (prodn (tag 'opt_actual_condition_parameters 'cp)
                     'empty))
      nil

  (if (rule m (prodn (tag 'opt_actual_condition_parameters 'cp)
                     (tag 'actual_condition_parameters 'cp2)))
      (actual_cargs (subtree m 'actual_condition_parameters))

  (if (rule m (prodn (tag 'actual_condition_parameters 'cp)
                     (list 'UNLESS 'OPEN_PAREN (tag 'opt_group_name 'g)
                           (tag 'identifier_list 'is) 'CLOSE_PAREN)))
      (id_list (subtree m 'identifier_list))

    nil)))))
```

```
  ( (lessp (tree_size m)) ))


(defn actual_dargs (m)

  (if (rule m (prodn (tag 'procedure_statement 's)
                     (list (tag 'IDENTIFIER 'pn)
                           (tag 'arg_list 'dp)
                           (tag 'opt_actual_condition_parameters 'cp))))
      (actual_dargs (subtree m 'arg_list))

  (if (rule m (prodn (tag 'arg_list 'as)
                     (list 'OPEN_PAREN (tag 'value_list 'vs)
                           'CLOSE_PAREN)))
      (actual_dargs (subtree m 'value_list))

  (if (rule m (prodn (tag 'value_list 'vs)
                     (tag 'expression 'e)))
      (rcons nil (subtree m 'expression))

  (if (rule m (prodn (tag 'value_list 'vs)
                     (list (tag 'value_list 'vs2)
                           'COMMA (tag 'expression 'e))))
      (rcons (actual_dargs (subtree m 'value_list))
             (subtree m 'expression))

    nil))))

  ( (lessp (tree_size m)) ))

(defn formal_cargs (u)

  (if (rule u (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (formal_cargs (subtree u 'opt_external_conditions))

  (if (rule u (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
                           'COLON (tag 'type_specification 'rt)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (formal_cargs (subtree u 'opt_external_conditions))

  (if (rule u (prodn (tag 'opt_external_conditions 'c)
                     'empty))
      nil

  (if (rule u (prodn (tag 'opt_external_conditions 'c)
                     (list 'UNLESS 'OPEN_PAREN 'COND
                           (tag 'identifier_list 'is) 'CLOSE_PAREN)))
      (id_list (subtree u 'identifier_list))

    nil))))

  ( (lessp (tree_size u)) ))


(defn exit_labels (e)

  (if (rule e (prodn (tag 'opt_exit_specification 'e)
                     'empty))
      nil

  (if (rule e (prodn (tag 'opt_exit_specification 'e)
                     (list 'EXIT
```

```
                           (tag 'non_validated_specification_expression
                                'se)
                           'SEMI_COLON)))
      nil

  (if (rule e (prodn (tag 'opt_exit_specification 'e)
                     (list 'EXIT (tag 'conditional_exit_specification 'c)
                           'SEMI_COLON)))
      (exit_labels (subtree e 'conditional_exit_specification))

  (if (rule e (prodn (tag 'conditional_exit_specification 'c)
                     (list 'CASE 'OPEN_PAREN (tag 'case_exit_body 'e)
                           'CLOSE_PAREN)))
      (exit_labels (subtree e 'case_exit_body))

  (if (rule e (prodn (tag 'case_exit_body 'b)
                     (tag 'case_exit 'c)))
      (exit_labels (subtree e 'case_exit))

  (if (rule e (prodn (tag 'case_exit_body 'b)
                     (list (tag 'case_exit_body 'b2) 'SEMI_COLON
                           (tag 'case_exit 'c))))
      (append (exit_labels (subtree e 'case_exit_body))
              (exit_labels (subtree e 'case_exit)))

  (if (rule e (prodn (tag 'case_exit 'ce)
                     (list 'IS (tag 'case_exit_labels 'l) 'COLON
                           (tag 'non_validated_specification_expression
                                'e))))
      (exit_labels (subtree e 'case_exit_labels))

  (if (rule e (prodn (tag 'case_exit_labels 'ls)
                     (list (tag 'case_exit_labels 'ls2) 'COMMA
                           (tag 'exit_label 'l))))
      (append (exit_labels (subtree e 'case_exit_labels))
              (exit_labels (subtree e 'exit_label)))

  (if (rule e (prodn (tag 'case_exit_labels 'ls)
                     (tag 'exit_label 'l)))
      (exit_labels (subtree e 'exit_label))

  (if (rule e (prodn (tag 'exit_label 'l)
                     (tag 'IDENTIFIER 'n)))
      (exit_labels (subtree e 'IDENTIFIER))

  (if (rule e (prodn (tag 'exit_label 'l)
                     'NORMAL))
      (list 'NORMAL)

  (if (identifierp e)
      (list (gname e))

    nil))))))))))))

  ( (lessp (tree_size e)) ))


(defn exit_spec (u)

  (if (rule u (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (exit_spec (subtree u 'procedure_body))

  (if (rule u (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
```

```
                          'COLON (tag 'type_specification 'rt)
                          (tag 'opt_external_conditions 'c)
                          'EQUAL (tag 'procedure_body 'b))))
      (exit_spec (subtree u 'procedure_body))

  (if (rule u (prodn (tag 'procedure_body 'b)
                    'PENDING))
      nil

  (if (rule u (prodn (tag 'procedure_body 'b)
                    (list 'BEGIN
                          (tag 'external_operational_specification 'es)
                          (tag 'opt_internal_environment 'iv)
                          (tag 'opt_keep_specification 'k)
                          (tag 'opt_internal_statements 'st)
                          'END)))
      (exit_spec (subtree u 'external_operational_specification))

  (if (rule u (prodn (tag 'external_operational_specification 's)
                    (list (tag 'opt_entry_specification 'e)
                          (tag 'opt_exit_specification 'x))))
      (subtree u 'opt_exit_specification)

  (if (rule u (prodn (tag 'opt_exit_specification 'e)
                    'empty))
      u

  (if (rule u (prodn (tag 'opt_exit_specification 'e)
                    (list 'EXIT
                          (tag 'non_validated_specification_expression
                               'se)
                          'SEMI_COLON)))
      u

  (if (rule u (prodn (tag 'opt_exit_specification 'e)
                    (list 'EXIT (tag 'conditional_exit_specification 'c)
                          'SEMI_COLON)))
      u

    nil))))))))

  ( (lessp (tree_size u)) ))


(defn handler (m c)

  (if (rule m (prodn (tag 'opt_condition_handlers 'c)
                    'empty))
      nil

  (if (rule m (prodn (tag 'opt_condition_handlers 'c)
                    (list 'WHEN (tag 'opt_handler_list 'hs))))
      (handler (subtree m 'opt_handler_list) c)


  (if (rule m (prodn (tag 'opt_handler_list 'hs)
                    'empty))
      nil

  (if (rule m (prodn (tag 'opt_handler_list 'hs)
                    (tag 'handler_list 'hs2)))
      (handler (subtree m 'handler_list) c)

  (if (rule m (prodn (tag 'handler_list 'hs)
                    (tag 'handler 'h)))
      (handler (subtree m 'handler) c)

  (if (rule m (prodn (tag 'handler_list 'hs)
                    (list (tag 'handler_list 'hs2)
```

```
                           (tag 'handler 'h))))
      (let ((r (handler (subtree m 'handler_list) c)))
        (if (equal r nil)
            (handler (subtree m 'handler) c)
          r))

  (if (rule m (prodn (tag 'handler 'h)
                     (list 'IS (tag 'identifier_list 'cs) 'COLON
                           (tag 'opt_internal_statements 's))))
      (if (member c (id_list (subtree m 'identifier_list)))
          (subtree m 'opt_internal_statements)
        nil)

    nil)))))))

  ( (lessp (tree_size m)) ))


(defn handler_labels (m)

  (if (rule m (prodn (tag 'opt_condition_handlers 'c)
                     'empty))
      nil

  (if (rule m (prodn (tag 'opt_condition_handlers 'c)
                     (list 'WHEN (tag 'opt_handler_list 'hs))))
      (handler_labels (subtree m 'opt_handler_list))


  (if (rule m (prodn (tag 'opt_handler_list 'hs)
                     'empty))
      nil

  (if (rule m (prodn (tag 'opt_handler_list 'hs)
                     (tag 'handler_list 'hs2)))
      (handler_labels (subtree m 'handler_list))

  (if (rule m (prodn (tag 'handler_list 'hs)
                     (tag 'handler 'h)))
      (handler_labels (subtree m 'handler))

  (if (rule m (prodn (tag 'handler_list 'hs)
                     (list (tag 'handler_list 'hs2)
                           (tag 'handler 'h))))
      (append (handler_labels (subtree m 'handler_list))
              (handler_labels (subtree m 'handler)))

  (if (rule m (prodn (tag 'handler 'h)
                     (list 'IS (tag 'identifier_list 'cs) 'COLON
                           (tag 'opt_internal_statements 's))))
      (id_list (subtree m 'identifier_list))

    nil)))))))

  ( (lessp (tree_size m)) ))


(defn internal_initial_value_exp (m)

  (if (rule m (prodn (tag 'internal_data_or_condition_objects 'iv)
                     (list (tag 'access_specification 'a)
                           (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ts)
                           (tag 'opt_internal_initial_value 'v)
                           'SEMI_COLON)))
      (internal_initial_value_exp (subtree m 'opt_internal_initial_value))

  (if (rule m (prodn (tag 'opt_internal_initial_value 'v)
                     'empty))
```

```
      nil

  (if (rule m (prodn (tag 'opt_internal_initial_value 'v)
                     (list 'COLON_EQUAL (tag 'expression 'e))))
      (subtree m 'expression)

    nil)))

  ( (lessp (tree_size m)) ))


(defn keep_spec (u)

  (if (rule u (prodn (tag 'procedure_declaration 'd)
                     (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                           (tag 'external_data_objects 'a)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (keep_spec (subtree u 'procedure_body))

  (if (rule u (prodn (tag 'function_declaration 'd)
                     (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                           (tag 'opt_external_data_objects 'a)
                           'COLON (tag 'type_specification 'rt)
                           (tag 'opt_external_conditions 'c)
                           'EQUAL (tag 'procedure_body 'b))))
      (keep_spec (subtree u 'procedure_body))

  (if (rule u (prodn (tag 'procedure_body 'b)
                     (list 'BEGIN
                           (tag 'external_operational_specification 'es)
                           (tag 'opt_internal_environment 'iv)
                           (tag 'opt_keep_specification 'k)
                           (tag 'opt_internal_statements 'st)
                           'END)))
      (keep_spec (subtree u 'opt_keep_specification))

  (if (rule u (prodn (tag 'opt_keep_specification 'k)
                     'empty))
      (mk_true_expression)

  (if (rule u (prodn (tag 'opt_keep_specification 'k)
                     (list 'KEEP
                           (tag 'non_validated_specification_expression 'se)
                           'SEMI_COLON)))
      (keep_spec (subtree u 'non_validated_specification_expression))

  (if (or (rule u (prodn (tag 'non_validated_specification_expression 'se)
                         (list 'OPEN_PAREN (tag 'proof_directive 'd)
                               (tag 'expression 'e) 'CLOSE_PAREN)))
          (rule u (prodn (tag 'non_validated_specification_expression 'se)
                         (list (tag 'proof_directive 'd)
                               (tag 'expression 'e))))
          (rule u (prodn (tag 'non_validated_specification_expression 'se)
                         (tag 'expression 'e))))
      (subtree u 'expression)

    nil))))))

  ( (lessp (tree_size u)) ))


(defn if_statement_else_part (s)

  (if (rule s (prodn (tag 'if_composition 's)
                     (list 'IF (tag 'expression 'b) 'THEN
                           (tag 'opt_internal_statements 'ss)
                           (tag 'if_composition_else_part 'ep)
                           (tag 'opt_condition_handlers 'cs) 'END)))
```

```
      (if_statement_else_part (subtree s 'if_composition_else_part))

  (if (rule s (prodn (tag 'if_composition_else_part 'ep)
                     'empty))
      nil

  (if (rule s (prodn (tag 'if_composition_else_part 'ep)
                     (list 'ELSE (tag 'opt_internal_statements 'ss))))
      (subtree s 'opt_internal_statements)

  (if (rule s (prodn (tag 'if_composition_else_part 'ep)
                     (list 'ELIF (tag 'expression 'b) 'THEN
                           (tag 'opt_internal_statements 'ss)
                           (tag 'if_composition_else_part 'ep2))))
      (mk_elif_into_if_statement s)

    nil))))

  ( (lessp (tree_size s)) ))


(defn new_name_arg (m)

  (if (rule m (prodn (tag 'move_statement 's)
                     (list 'MOVE (tag 'removable_component 'c)
                           (tag 'component_destination 'd))))
      (new_name_arg (subtree m 'component_destination))

  (if (rule m (prodn (tag 'component_destination 'd)
                     (tag 'new_dynamic_variable_component 'dc)))
      (new_name_arg (subtree m 'new_dynamic_variable_component))

  (if (rule m (prodn (tag 'component_destination 'd)
                     (list 'TO (tag 'name_expression 'ne))))
      (subtree m 'name_expression)

  (if (rule m (prodn (tag 'new_statement 's)
                     (list 'NEW (tag 'expression 'e)
                           (tag 'new_dynamic_variable_component 'dc))))
      (new_name_arg (subtree m 'new_dynamic_variable_component))

  (if (or (rule m (prodn (tag 'new_dynamic_variable_component 'dc)
                         (list 'INTO (tag 'name_expression 'ne))))
          (rule m (prodn (tag 'new_dynamic_variable_component 'dc)
                         (list 'INTO 'SET (tag 'name_expression 'ne))))
          (rule m (prodn (tag 'new_dynamic_variable_component 'dc)
                         (list 'BEFORE (tag 'name_expression 'ne))))
          (rule m (prodn (tag 'new_dynamic_variable_component 'dc)
                         (list 'BEFORE 'SEQ (tag 'name_expression 'ne))))
          (rule m (prodn (tag 'new_dynamic_variable_component 'dc)
                         (list 'BEHIND (tag 'name_expression 'ne))))
          (rule m (prodn (tag 'new_dynamic_variable_component 'dc)
                         (list 'BEHIND 'SEQ (tag 'name_expression 'ne)))))
      (subtree m 'name_expression)

    nil)))))

  ( (lessp (tree_size m)) ))


(defn remove_exp_arg (m)

  (if (rule m (prodn (tag 'move_statement 's)
                     (list 'MOVE (tag 'removable_component 'c)
                           (tag 'component_destination 'd))))
      (remove_exp_arg (subtree m 'removable_component))

  (if (rule m (prodn (tag 'remove_statement 's)
                     (list 'REMOVE (tag 'removable_component 'c))))
```

```
      (remove_exp_arg (subtree m 'removable_component))

  (if (rule m (prodn (tag 'removable_component 'c)
                     (list 'ELEMENT (tag 'expression 'e)
                           'FROM 'SET (tag 'name_expression 'ne))))
      (subtree m 'expression)

  (if (rule m (prodn (tag 'removable_component 'c)
                     (tag 'name_expression 'e)))
      nil

    nil))))

  ( (lessp (tree_size m)) ))


(defn remove_name_arg (m)

  (if (rule m (prodn (tag 'move_statement 's)
                     (list 'MOVE (tag 'removable_component 'c)
                           (tag 'component_destination 'd))))
      (remove_name_arg (subtree m 'removable_component))

  (if (rule m (prodn (tag 'remove_statement 's)
                     (list 'REMOVE (tag 'removable_component 'c))))
      (remove_name_arg (subtree m 'removable_component))

  (if (rule m (prodn (tag 'removable_component 'c)
                     (list 'ELEMENT (tag 'expression 'e)
                           'FROM 'SET (tag 'name_expression 'ne))))
      (subtree m 'name_expression)

  (if (rule m (prodn (tag 'removable_component 'c)
                     (tag 'name_expression 'e)))
      (subtree m 'name_expression)

    nil))))

  ( (lessp (tree_size m)) ))


(defn procedure_body (d)

  (if (or (rule d (prodn (tag 'procedure_declaration 'd)
                         (list 'PROCEDURE (tag 'IDENTIFIER 'pn)
                               (tag 'external_data_objects 'a)
                               (tag 'opt_external_conditions 'c)
                               'EQUAL (tag 'procedure_body 'b))))
          (rule d (prodn (tag 'function_declaration 'd)
                         (list 'FUNCTION (tag 'IDENTIFIER 'fn)
                               (tag 'opt_external_data_objects 'a)
                               'COLON (tag 'type_specification 'rt)
                               (tag 'opt_external_conditions 'c)
                               'EQUAL (tag 'procedure_body 'b)))))
      (subtree d 'procedure_body)

    nil))


; ****************************************************************************
;
;                                  THE STATE
;
; ****************************************************************************

; The state is a marked object <mark , map>.  The mark is either NIL or 'IND,
; which indicates an indeterminate state.  The map is a name-value mapping.
; It maps variables (and constants) to their (marked typed) values.  In
```

```
; addition, it contains the special components:
;
;     entry - the (marked typed) value resulting from evaluation of the
;             entry specification on the initial state
;     exit - the (marked typed) value resulting from evaluation of the
;            exit specification on the final state
;     keep - the (marked typed) value that is the conjunction of results
;            from all evaluations of the keep specification
;     assert - the (marked typed) value that is the conjunction of results
;              from all evaluations of assert specifications
;     keep~ - the parse tree for the keep specification
;     cond~ - the currently active condition
;     result~ - the (marked typed) value resulting from expression evaluation
;     var - a name-value mapping from variable names, arguments and locals,
;           to local|formal
;     const - a name-value mapping from constant names, arguments and locals,
;             to local|formal
;     cond - a name-value mapping from condition names, arguments and locals,
;            to local|formal

(defn default_state ()
  (marked nil
          (list (cons 'entry (Gtrue))
                (cons 'exit (Gtrue))
                (cons 'keep (Gtrue))
                (cons 'assert (Gtrue))
                (cons 'keep~ (mk_true_expression))
                (cons 'cond~ 'normal)
                (cons 'result~ (Gizero))
                (cons 'var nil)
                (cons 'const nil)
                (cons 'cond '((routineerror . formal)
                              (spaceerror . formal))))))


; ========================================
;  Extraction of Components from the State
; ========================================

(defn map (s)
  (object s))

(defn state_componentp (k s)
  (in_map (map s) k))

(defn state_component (k s)
  (mapped_value (map s) k))

(defn entry (s)
  (mapped_value (map s) 'entry))

(defn exit (s)
  (mapped_value (map s) 'exit))

(defn keep (s)
  (mapped_value (map s) 'keep))

(defn assert (s)
  (mapped_value (map s) 'assert))

(defn keep~ (s)
  (mapped_value (map s) 'keep~))

(defn cond~ (s)
  (mapped_value (map s) 'cond~))

(defn condition_normal (s)
  (equal (cond~ s) 'normal))
```

```
(defn condition_non_normal (s)
  (not (condition_normal s)))

(defn normal_state (s)
  (and (determinate s)
       (condition_normal s)))

(defn result~ (s)
  (mapped_value (map s) 'result~))

(defn result~_list (ss)
  (if (nlistp ss)
      nil
    (cons (result~ (car ss))
          (result~_list (cdr ss)))))

(defn var (s)
  (mapped_value (map s) 'var))

(defn const (s)
  (mapped_value (map s) 'const))

(defn cond+ (s)
  (mapped_value (map s) 'cond))

(defn variablep (id s)
  (and (state_componentp id s)
       (in_map (var s) id)))

(defn conditionp (id s)
  (in_map (cond+ s) id))

(defn all_conditionsp (ids s)
  (if (nlistp ids)
      T
    (and (conditionp (car ids) s)
         (all_conditionsp (cdr ids) s))))

(defn type_of (n s)
  (type (state_component n s)))

(defn mode (td)
  (root td))

(defn locals (cs)
  (if (nlistp cs)
      nil
  (if (equal (cdar cs) 'local)
      (cons (car cs) (locals (cdr cs)))
    (locals (cdr cs)))))

(defn local_vars (s)
  (locals (var s)))

(defn local_consts (s)
  (locals (const s)))

(defn local_conds (s)
  (locals (cond+ s)))


; =================================
;  Setting Components of the State
; =================================

(defn mark_state_indeterminate (s)
  (marked 'ind (map s)))

(defn store_value (k v s)
```

```
  (marked (mark s)
          (add_to_map (map s) k v)))

(defn set_condition (s c)
  ; Set the 'cond~ component of the map from state s with the value c.
  (store_value 'cond~ c s))

(defn store_result~ (v s)
  (if (determinate v)
      (store_value 'result~ v s)
    (set_condition s 'routineerror)))

(defn all_determinate (vs)
  (if (nlistp vs)
      (equal vs nil)
    (and (determinate (car vs))
         (all_determinate (cdr vs)))))

(defn reset_leave_to_normal (s)
  (if (equal (cond~ s) 'leave)
      (set_condition s 'normal)
    s))

(defn record_assertion (a s)
  (store_value 'assert
               (Gand (assert s) a)
               s))

(defn store_cond (id sc s)
  ; Adds a new condition to known conditions in s.
  ; id is the name of the condition
  ; sc is 'formal or 'local
  ; s is the state
  (store_value 'cond
               (cons (cons id sc) (cond+ s))
               s))

(defn note_conds (cs sc s)
  ; cs is a list of condition names
  ; sc is 'formal or 'local
  ; s is the state
  (if (nlistp cs)
      s
    (store_cond (car cs) sc
                (note_conds (cdr cs) sc s))))

(defn store_const (id v sc s)
  ; Adds a new constant to s.
  ; id is the name of the constant
  ; v is id's value
  ; sc is 'formal or 'local
  ; s is the state
  (store_value 'const
               (cons (cons id sc) (const s))
               (store_value id v s)))

(defn store_var (id v sc s)
  ; Adds a new variable to s.
  ; id is the name of the variable
  ; v is id's value
  ; sc is 'formal or 'local
  ; s is the state
  (store_value 'var
               (cons (cons id sc) (var s))
               (store_value id v s)))


; ====================================
;  Deleting Components from the State
```

```
; ====================================

(defn deallocate (k s)
  (marked (mark s)
          (remove (assoc k (map s)) (map s))))

(defn deallocate_vars (vs s)
  (if (nlistp vs)
      s
    (deallocate_vars (cdr vs)
                     (deallocate (caar vs)
                                 (store_value 'var
                                              (remove (car vs) (var s))
                                              s)))))

(defn deallocate_consts (cs s)
  (if (nlistp cs)
      s
    (deallocate_consts (cdr cs)
                       (deallocate (caar cs)
                                   (store_value 'const
                                                (remove (car cs) (const s))
                                                s)))))

(defn deallocate_conds (cs s)
  (if (nlistp cs)
      s
    (deallocate_conds (cdr cs)
                      (store_value 'cond
                                   (remove (car cs) (cond+ s))
                                   s))))


; ================
;  State Equality
; ================

(defn scomp_equal (k v1 v2)
  (if (member k '(var const cond))
      (set_equal v1 v2)
  (if (equal k 'cond~)
      (equal v1 v2)
  (if (equal k 'keep~)
      (tree_equal v1 v2)
    (and (Gtruep (Gequal v1 v2))
         (equal (type v1) (type v2)))))))

(defn ssubmap (m1 m2)
  (if (nlistp m1)
      T
    (and (in_map m2 (caar m1))
         (scomp_equal (caar m1) (cdar m1)
                      (mapped_value m2 (caar m1)))
         (ssubmap (cdr m1) m2))))

(defn smap_equal (m1 m2)
  (and (ssubmap m1 m2)
       (ssubmap m2 m1)))

(defn sequal (s1 s2)
  (if (equal s1 s2)
      T
    (and (equal (mark s1) (mark s2))
         (smap_equal (map s1) (map s2)))))



; ********************
;  Constraint Support
```

```
; ********************

(defn-sk+ implementation_constrained (s1 s0)
  ; s1 is the updated state when a condition is signalled
  ; s0 is the state before update
  (forall k (implies (not (equal k 'cond~))
                     (equal (mapped_value (map s1) k)
                            (mapped_value (map s0) k)))))

(defn state_check (ss s0)
  ; ss is a list of states resulting from evaluation of an argument list in
  ; state s0
  (if (or (nlistp ss) (not (normal_state s0)))
      s0
  (if (normal_state (car ss))
      (state_check (cdr ss) s0)
    (set_condition (marked (mark (car ss)) (map s0))
                   (cond~ (car ss))))))

(disable sequal)
(disable store_value)
(disable cond~)
(enable determinate)


; ****************
;  Literal Values
; ****************

(constrain p_Gfalse_intro (rewrite)
  (let ((s1 (p_Gfalse s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gfalse) s0))
                       (determinate (Gfalse))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gfalse (lambda (s0)
                (mark_state_indeterminate s0))) ))

(defn GPF_false (s0)
  (if (normal_state s0)
      (p_Gfalse s0)
    s0))


(constrain p_Gtrue_intro (rewrite)
  (let ((s1 (p_Gtrue s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gtrue) s0))
                       (determinate (Gtrue))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gtrue (lambda (s0)
               (mark_state_indeterminate s0))) ))

(defn GPF_true (s0)
  (if (normal_state s0)
      (p_Gtrue s0)
    s0))

(constrain p_minteger_intro (rewrite)
```

```
  (let ((s1 (p_minteger e s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) ’normal))
                  (and (sequal s1
                               (store_value ’result~ (minteger e) s0))
                       (determinate (minteger e))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) ’normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) ’(routineerror spaceerror))))))
  ( (p_minteger (lambda (e s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_minteger (e s)
  (if (normal_state s)
      (p_minteger e s)
    s))

(constrain p_Gchar_intro (rewrite)
  (let ((s1 (p_Gchar e s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) ’normal))
                  (and (sequal s1
                               (store_value ’result~ (Gchar e) s0))
                       (determinate (Gchar e))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) ’normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) ’(routineerror spaceerror))))))
  ( (p_Gchar (lambda (e s0)
               (mark_state_indeterminate s0))) ))

(defn GPF_Gchar (e s)
  (if (normal_state s)
      (p_Gchar e s)
    s))

(constrain p_Gstring_seq_intro (rewrite)
  (let ((s1 (p_Gstring_seq e s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) ’normal))
                  (and (sequal s1
                               (store_value ’result~ (Gstring_seq e) s0))
                       (determinate (Gstring_seq e))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) ’normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) ’(routineerror spaceerror))))))
  ( (p_Gstring_seq (lambda (e s0)
                     (mark_state_indeterminate s0))) ))

(defn GPF_Gstring_seq (e s)
  (if (normal_state s)
      (p_Gstring_seq e s)
    s))


; *****************
;  Gypsy Operations
; *****************

; --------------------
;  Component Selection
; --------------------

(disable array_get)

(constrain p_array_get_intro (rewrite)
```

```
  (let ((s1 (p_array_get a i td s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (array_get a i td) s0))
                       (determinate (array_get a i td))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_array_get (lambda (a i td s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_array_get (sA sI s0)
  (let ((r (state_check (list sA sI) s0)))
    (if (normal_state r)
        (p_array_get (result~ sA) (result~ sI)
                     (type (result~ sA)) s0)
      r)))


(disable mapping_get)

(constrain p_mapping_get_intro (rewrite)
  (let ((s1 (p_mapping_get m d td s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (mapping_get m d td) s0))
                       (determinate (mapping_get m d td))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_mapping_get (lambda (m d td s0)
                     (mark_state_indeterminate s0))) ))

(defn GPF_mapping_get (sM sD s0)
  (let ((r (state_check (list sM sD) s0)))
    (if (normal_state r)
        (p_mapping_get (result~ sM) (result~ sD)
                       (type (result~ sM)) s0)
      r)))


(disable record_get)

(constrain p_record_get_intro (rewrite)
  (let ((s1 (p_record_get r fn td s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (record_get r fn td) s0))
                       (determinate (record_get r fn td))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_record_get (lambda (r fn td s0)
                    (mark_state_indeterminate s0))) ))

(defn GPF_record_get (sR sFn s0)
  (let ((r (state_check (list sR sFn) s0)))
    (if (normal_state r)
        (p_record_get (result~ sR) (result~ sFn)
                      (type (result~ sR)) s0)
      r)))
```

```
(disable sequence_get)

(constrain p_sequence_get_intro (rewrite)
  (let ((s1 (p_sequence_get s i td s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (sequence_get s i td)
                                            s0))
                       (determinate (sequence_get s i td))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_sequence_get (lambda (s i td s0)
                      (mark_state_indeterminate s0))) ))

(defn GPF_sequence_get (sS sI s0)
  (let ((r (state_check (list sS sI) s0)))
    (if (normal_state r)
        (p_sequence_get (result~ sS) (result~ sI)
                        (type (result~ sS)) s0)
      r)))


(disable state_check)
(disable normal_state)
(disable mode)
(disable type)
(disable result~)
(disable GPF_array_get)
(disable GPF_record_get)
(disable GPF_mapping_get)
(disable GPF_sequence_get)
(disable store_result~)
(disable not_selectable_error)
(disable default_value)
(disable integer_desc)
(disable *1*integer_desc)

;; >> It might be better to just constrain this and forget the four defn's
;; above.

(defn GPF_select_op (sV sS s0)
  ; sV is the state whose result~ component is the parent value
  ; sS is a list of states resulting from evaluation of a selector list
  ; s0 is the state in which the selection is to be done
  (let ((r (state_check (cons sV sS) s0)))
    (if (not (normal_state r))
        r
    (if (nlistp sS)
        sV
      (case (mode (type (result~ sV)))
        (array (GPF_select_op (GPF_array_get sV (car sS) s0)
                              (cdr sS) s0))
        (record (GPF_select_op (GPF_record_get sV (car sS) s0)
                               (cdr sS) s0))
        (mapping (GPF_select_op (GPF_mapping_get sV (car sS) s0)
                                (cdr sS) s0))
        (sequence (GPF_select_op (GPF_sequence_get sV (car sS) s0)
                                 (cdr sS) s0))
        (otherwise (store_result~ (marked (not_selectable_error (result~ sV))
                                          (default_value (integer_desc)))
                                  s0))))))
  ( (lessp (count sS)) ))


; ----------------------
;  Subsequence Selection
```

```
; ----------------------

(disable subsequence_get)

(constrain p_subsequence_get_intro (rewrite)
  (let ((s1 (p_subsequence_get s lo hi s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (subsequence_get s lo hi)
                                            s0))
                       (determinate (subsequence_get s lo hi))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_subsequence_get (lambda (s lo hi s0)
                         (mark_state_indeterminate s0))) ))

(defn GPF_subsequence_get (sS sLO sHI s0)
  (let ((r (state_check (list sS sLO sHI) s0)))
    (if (normal_state r)
        (p_subsequence_get (result~ sS) (result~ sLO) (result~ sHI) s0)
      r)))


; -----------------
;  Value Alteration
; -----------------

(disable array_put)

(constrain p_array_put_intro (rewrite)
  (let ((s1 (p_array_put a i v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (array_put a i v) s0))
                       (determinate (array_put a i v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_array_put (lambda (a i v s0)
                   (mark_state_indeterminate s0))) ))

(defn GPF_array_put (sA sI sV s0)
  (let ((r (state_check (list sA sI sV) s0)))
    (if (normal_state r)
        (p_array_put (result~ sA) (result~ sI) (result~ sV) s0)
      r)))


(disable record_put)

(constrain p_record_put_intro (rewrite)
  (let ((s1 (p_record_put r fn v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (record_put r fn v) s0))
                       (determinate (record_put r fn v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_record_put (lambda (r fn v s0)
                    (mark_state_indeterminate s0))) ))
```

```
(defn GPF_record_put (sR sFn sV s0)
  (let ((r (state_check (list sR sFn sV) s0)))
    (if (normal_state r)
        (p_record_put (result~ sR) (result~ sFn) (result~ sV) s0)
      r)))


(disable mapping_put)

(constrain p_mapping_put_intro (rewrite)
  (let ((s1 (p_mapping_put m d v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (mapping_put m d v) s0))
                       (determinate (mapping_put m d v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_mapping_put (lambda (m d v s0)
                     (mark_state_indeterminate s0))) ))

(defn GPF_mapping_put (sM sD sV s0)
  (let ((r (state_check (list sM sD sV) s0)))
    (if (normal_state r)
        (p_mapping_put (result~ sM) (result~ sD) (result~ sV) s0)
      r)))


(disable sequence_put)

(constrain p_sequence_put_intro (rewrite)
  (let ((s1 (p_sequence_put s i v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (sequence_put s i v) s0))
                       (determinate (sequence_put s i v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_sequence_put (lambda (s i v s0)
                      (mark_state_indeterminate s0))) ))

(defn GPF_sequence_put (sS sI sV s0)
  (let ((r (state_check (list sS sI sV) s0)))
    (if (normal_state r)
        (p_sequence_put (result~ sS) (result~ sI) (result~ sV) s0)
      r)))


(defn GPF_put_op (sBV sS sV s0)
  ; sBV is the state whose result~ component is the base (marked typed) value,
  ; sS is a list of states resulting from evaluation of the component
  ;    selectors,
  ; sV is the state whose result~ component is the new value for the selected
  ;    component
  ; s0 is the starting state for the put
  ; Example Gypsy:  z with ([i,j,k].f1[l][m].f2 := e)
  (let ((r (state_check (cons sBV (rcons sS sV)) s0)))
    (if (not (normal_state r))
        r
    (if (nlistp sS)
        (store_result~ (result~ sV) s0)
      (case (mode (type (result~ sBV)))
        (array (GPF_array_put sBV (car sS)
                              (GPF_put_op (GPF_array_get sBV (car sS) s0)
```

```
                                          (cdr sS) sV s0)
                              s0))
        (record (GPF_record_put sBV (car sS)
                                (GPF_put_op (GPF_record_get sBV (car sS) s0)
                                            (cdr sS) sV s0)
                                s0))
        (mapping (GPF_mapping_put sBV (car sS)
                                  (GPF_put_op (GPF_mapping_get sBV (car sS) s0)
                                              (cdr sS) sV s0)
                                  s0))
        (sequence (GPF_sequence_put sBV (car sS)
                                    (GPF_put_op (GPF_sequence_get sBV (car sS)
                                                                  s0)
                                                (cdr sS) sV s0)
                                    s0))
        (otherwise (store_result~
                     (marked (component_assign_error (result~ sBV))
                             (default_value (integer_desc)))
                     s0)))))))


(disable Gmapomit)

(constrain p_Gmapomit_intro (rewrite)
  (let ((s1 (p_Gmapomit m i s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gmapomit m i) s0))
                       (determinate (Gmapomit m i))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gmapomit (lambda (m i s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_Gmapomit (sM sI s0)
  (let ((r (state_check (list sM sI) s0)))
    (if (normal_state r)
        (p_Gmapomit (result~ sM) (result~ sI) s0)
      r)))


(disable Gseqomit)

(constrain p_Gseqomit_intro (rewrite)
  (let ((s1 (p_Gseqomit s i s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gseqomit s i) s0))
                       (determinate (Gseqomit s i))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gseqomit (lambda (s i s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_Gseqomit (sS sI s0)
  (let ((r (state_check (list sS sI) s0)))
    (if (normal_state r)
        (p_Gseqomit (result~ sS) (result~ sI) s0)
      r)))


(disable Gmap_insert)
```

```
(constrain p_Gmap_insert_intro (rewrite)
  (let ((s1 (p_Gmap_insert m d v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gmap_insert m d v) s0))
                       (determinate (Gmap_insert m d v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gmap_insert (lambda (m d v s0)
                     (mark_state_indeterminate s0))) ))

(defn GPF_Gmap_insert (sM sD sV s0)
  (let ((r (state_check (list sM sD sV) s0)))
    (if (normal_state r)
        (p_Gmap_insert (result~ sM) (result~ sD) (result~ sV) s0)
      r)))


(disable Gseq_insert_before)

(constrain p_Gseq_insert_before_intro (rewrite)
  (let ((s1 (p_Gseq_insert_before s i v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~
                                            (Gseq_insert_before s i v)
                                            s0))
                       (determinate (Gseq_insert_before s i v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gseq_insert_before (lambda (s i v s0)
                            (mark_state_indeterminate s0))) ))

(defn GPF_Gseq_insert_before (sS sI sV s0)
  (let ((r (state_check (list sS sI sV) s0)))
    (if (normal_state r)
        (p_Gseq_insert_before (result~ sS) (result~ sI) (result~ sV) s0)
      r)))


(disable Gseq_insert_behind)

(constrain p_Gseq_insert_behind_intro (rewrite)
  (let ((s1 (p_Gseq_insert_behind s i v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~
                                            (Gseq_insert_behind s i v)
                                            s0))
                       (determinate (Gseq_insert_behind s i v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gseq_insert_behind (lambda (s i v s0)
                            (mark_state_indeterminate s0))) ))

(defn GPF_Gseq_insert_behind (sS sI sV s0)
  (let ((r (state_check (list sS sI sV) s0)))
    (if (normal_state r)
        (p_Gseq_insert_behind (result~ sS) (result~ sI) (result~ sV) s0)
      r)))
```

```
; --------------------------
;  Sequence/Set Constructors
; --------------------------

(disable Gseq)

(constrain p_Gseq_intro (rewrite)
  (let ((s1 (p_Gseq es td s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gseq es td) s0))
                       (determinate (Gseq es td))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gseq (lambda (es td s0)
              (mark_state_indeterminate s0))) ))

(defn GPF_Gseq (sES td s0)
  (let ((r (state_check sES s0)))
    (if (normal_state r)
        (p_Gseq (result~_list sES) td s0)
      r)))


(disable Gset)

(constrain p_Gset_intro (rewrite)
  (let ((s1 (p_Gset es td s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gset es td) s0))
                       (determinate (Gset es td))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gset (lambda (es td s0)
              (mark_state_indeterminate s0))) ))

(defn GPF_Gset (sES td s0)
  (let ((r (state_check sES s0)))
    (if (normal_state r)
        (p_Gset (result~_list sES) td s0)
      r)))


(disable Grange_elements)

(constrain p_Grange_elements_intro (rewrite)
  (let ((s1 (p_Grange_elements lo hi s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Grange_elements lo hi)
                                            s0))
                       (all_determinate (Grange_elements lo hi))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Grange_elements (lambda (lo hi s0)
                         (mark_state_indeterminate s0))) ))

(defn range_element_state_list2 (es s)
  (if (nlistp es)
```

```
      nil
    (cons (store_result~ (car es) s)
          (range_element_state_list2 (cdr es) s))))

(defn range_element_state_list (s)
  (if (normal_state s)
      (range_element_state_list2 (result~ s) s)
    (list s)))

(defn GPF_Grange_elements (sLO sHI s0)
  (let ((r (state_check (list sLO sHI) s0)))
    (if (normal_state r)
        (range_element_state_list
         (p_Grange_elements (result~ sLO) (result~ sHI) s0))
      r)))


(defn GPF_Gset_or_seq (m sES td s0)
  (if (rule m (prodn (tag 'set_or_seq_mark 'm)
                     (list 'SET 'COLON)))
      (GPF_Gset sES td s0)
  (if (rule m (prodn (tag 'set_or_seq_mark 'm)
                     (list 'SEQ 'COLON)))
      (GPF_Gseq sES td s0)
    (GPF_Gseq sES td s0))))



; ----------------
;  Unary Operators
; ----------------

(disable Gminus)

(constrain p_Gminus_intro (rewrite)
  (let ((s1 (p_Gminus v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gminus v) s0))
                       (determinate (Gminus v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gminus (lambda (v s0)
                (mark_state_indeterminate s0))) ))

(defn GPF_Gminus (sV s0)
  (let ((r (state_check (list sV) s0)))
    (if (normal_state r)
        (p_Gminus (result~ sV) s0)
      r)))


(disable Gnot)

(constrain p_Gnot_intro (rewrite)
  (let ((s1 (p_Gnot v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gnot v) s0))
                       (determinate (Gnot v))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gnot (lambda (v s0)
```

```
              (mark_state_indeterminate s0))) ))

(defn GPF_Gnot (sV s0)
  (let ((r (state_check (list sV) s0)))
    (if (normal_state r)
        (p_Gnot (result~ sV) s0)
      r)))


(defn GPF_apply_unary_op (op sV s0)
  (if (rule op (prodn (tag 'unary_operator 'op) 'MINUS))
      (GPF_Gminus sV s0)

  (if (rule op (prodn (tag 'unary_operator 'op) 'NOT))
      (GPF_Gnot sV s0)

    (mark_state_indeterminate s0))))



; ------------------
;  Binary Operators
; ------------------

(disable Gequal)

(constrain p_Gequal_intro (rewrite)
  (let ((s1 (p_Gequal v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gequal v1 v2) s0))
                       (determinate (Gequal v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gequal (lambda (v1 v2 s0)
                (mark_state_indeterminate s0))) ))

(defn GPF_Gequal (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gequal (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gne)

(constrain p_Gne_intro (rewrite)
  (let ((s1 (p_Gne v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gne v1 v2) s0))
                       (determinate (Gne v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gne (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Gne (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gne (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Glt)

(constrain p_Glt_intro (rewrite)
  (let ((s1 (p_Glt v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Glt v1 v2) s0))
                       (determinate (Glt v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Glt (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Glt (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Glt (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gor)

(constrain p_Gor_intro (rewrite)
  (let ((s1 (p_Gor v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gor v1 v2) s0))
                       (determinate (Gor v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gor (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Gor (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gor (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gle)

(constrain p_Gle_intro (rewrite)
  (let ((s1 (p_Gle v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gle v1 v2) s0))
                       (determinate (Gle v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gle (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Gle (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gle (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Ggt)

(constrain p_Ggt_intro (rewrite)
  (let ((s1 (p_Ggt v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Ggt v1 v2) s0))
                       (determinate (Ggt v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Ggt (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Ggt (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Ggt (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gge)

(constrain p_Gge_intro (rewrite)
  (let ((s1 (p_Gge v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gge v1 v2) s0))
                       (determinate (Gge v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gge (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Gge (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gge (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gand)

(constrain p_Gand_intro (rewrite)
  (let ((s1 (p_Gand v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gand v1 v2) s0))
                       (determinate (Gand v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gand (lambda (v1 v2 s0)
              (mark_state_indeterminate s0))) ))

(defn GPF_Gand (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gand (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Gimp)

(constrain p_Gimp_intro (rewrite)
  (let ((s1 (p_Gimp v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gimp v1 v2) s0))
                       (determinate (Gimp v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gimp (lambda (v1 v2 s0)
              (mark_state_indeterminate s0))) ))

(defn GPF_Gimp (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gimp (result~ sV1) (result~ sV2) s0)
      r)))


(disable Giff)

(constrain p_Giff_intro (rewrite)
  (let ((s1 (p_Giff v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Giff v1 v2) s0))
                       (determinate (Giff v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Giff (lambda (v1 v2 s0)
              (mark_state_indeterminate s0))) ))

(defn GPF_Giff (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Giff (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gpower)

(constrain p_Gpower_intro (rewrite)
  (let ((s1 (p_Gpower v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gpower v1 v2) s0))
                       (determinate (Gpower v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gpower (lambda (v1 v2 s0)
                (mark_state_indeterminate s0))) ))

(defn GPF_Gpower (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gpower (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Gtimes)

(constrain p_Gtimes_intro (rewrite)
  (let ((s1 (p_Gtimes v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gtimes v1 v2) s0))
                       (determinate (Gtimes v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gtimes (lambda (v1 v2 s0)
                (mark_state_indeterminate s0))) ))

(defn GPF_Gtimes (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gtimes (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gquotient)

(constrain p_Gquotient_intro (rewrite)
  (let ((s1 (p_Gquotient v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gquotient v1 v2) s0))
                       (determinate (Gquotient v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gquotient (lambda (v1 v2 s0)
                   (mark_state_indeterminate s0))) ))

(defn GPF_Gquotient (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gquotient (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gdiv)

(constrain p_Gdiv_intro (rewrite)
  (let ((s1 (p_Gdiv v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gdiv v1 v2) s0))
                       (determinate (Gdiv v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gdiv (lambda (v1 v2 s0)
              (mark_state_indeterminate s0))) ))

(defn GPF_Gdiv (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gdiv (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Gmod)

(constrain p_Gmod_intro (rewrite)
  (let ((s1 (p_Gmod v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) ’normal))
                  (and (sequal s1
                               (store_value ’result~ (Gmod v1 v2) s0))
                       (determinate (Gmod v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) ’normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) ’(routineerror spaceerror))))))
  ( (p_Gmod (lambda (v1 v2 s0)
            (mark_state_indeterminate s0))) ))

(defn GPF_Gmod (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gmod (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gplus)

(constrain p_Gplus_intro (rewrite)
  (let ((s1 (p_Gplus v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) ’normal))
                  (and (sequal s1
                               (store_value ’result~ (Gplus v1 v2) s0))
                       (determinate (Gplus v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) ’normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) ’(routineerror spaceerror))))))
  ( (p_Gplus (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Gplus (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gplus (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gsubtract)

(constrain p_Gsubtract_intro (rewrite)
  (let ((s1 (p_Gsubtract v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) ’normal))
                  (and (sequal s1
                               (store_value ’result~ (Gsubtract v1 v2) s0))
                       (determinate (Gsubtract v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) ’normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) ’(routineerror spaceerror))))))
  ( (p_Gsubtract (lambda (v1 v2 s0)
                 (mark_state_indeterminate s0))) ))

(defn GPF_Gsubtract (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gsubtract (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Gin)

(constrain p_Gin_intro (rewrite)
  (let ((s1 (p_Gin v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gin v1 v2) s0))
                       (determinate (Gin v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gin (lambda (v1 v2 s0)
             (mark_state_indeterminate s0))) ))

(defn GPF_Gin (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gin (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gunion)

(constrain p_Gunion_intro (rewrite)
  (let ((s1 (p_Gunion v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gunion v1 v2) s0))
                       (determinate (Gunion v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gunion (lambda (v1 v2 s0)
                (mark_state_indeterminate s0))) ))

(defn GPF_Gunion (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gunion (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gadjoin)

(constrain p_Gadjoin_intro (rewrite)
  (let ((s1 (p_Gadjoin v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gadjoin v1 v2) s0))
                       (determinate (Gadjoin v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gadjoin (lambda (v1 v2 s0)
                 (mark_state_indeterminate s0))) ))

(defn GPF_Gadjoin (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gadjoin (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Gomit)

(constrain p_Gomit_intro (rewrite)
  (let ((s1 (p_Gomit v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gomit v1 v2) s0))
                       (determinate (Gomit v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gomit (lambda (v1 v2 s0)
               (mark_state_indeterminate s0))) ))

(defn GPF_Gomit (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gomit (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gsub)

(constrain p_Gsub_intro (rewrite)
  (let ((s1 (p_Gsub v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gsub v1 v2) s0))
                       (determinate (Gsub v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gsub (lambda (v1 v2 s0)
              (mark_state_indeterminate s0))) ))

(defn GPF_Gsub (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gsub (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gintersect)

(constrain p_Gintersect_intro (rewrite)
  (let ((s1 (p_Gintersect v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gintersect v1 v2) s0))
                       (determinate (Gintersect v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gintersect (lambda (v1 v2 s0)
                    (mark_state_indeterminate s0))) ))

(defn GPF_Gintersect (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gintersect (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Gdifference)

(constrain p_Gdifference_intro (rewrite)
  (let ((s1 (p_Gdifference v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gdifference v1 v2) s0))
                       (determinate (Gdifference v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gdifference (lambda (v1 v2 s0)
                     (mark_state_indeterminate s0))) ))

(defn GPF_Gdifference (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gdifference (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gappend)

(constrain p_Gappend_intro (rewrite)
  (let ((s1 (p_Gappend v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gappend v1 v2) s0))
                       (determinate (Gappend v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gappend (lambda (v1 v2 s0)
                 (mark_state_indeterminate s0))) ))

(defn GPF_Gappend (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gappend (result~ sV1) (result~ sV2) s0)
      r)))


(disable Gcons)

(constrain p_Gcons_intro (rewrite)
  (let ((s1 (p_Gcons v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Gcons v1 v2) s0))
                       (determinate (Gcons v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Gcons (lambda (v1 v2 s0)
               (mark_state_indeterminate s0))) ))

(defn GPF_Gcons (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Gcons (result~ sV1) (result~ sV2) s0)
      r)))
```

```
(disable Grcons)

(constrain p_Grcons_intro (rewrite)
  (let ((s1 (p_Grcons v1 v2 s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (Grcons v1 v2) s0))
                       (determinate (Grcons v1 v2))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_Grcons (lambda (v1 v2 s0)
                (mark_state_indeterminate s0))) ))

(defn GPF_Grcons (sV1 sV2 s0)
  (let ((r (state_check (list sV1 sV2) s0)))
    (if (normal_state r)
        (p_Grcons (result~ sV1) (result~ sV2) s0)
      r)))


(defn GPF_apply_binary_op (op sV1 sV2 s0)

  (if (or (rule op (prodn (tag 'binary_operator 'op) 'EQ))
          (rule op (prodn (tag 'binary_operator 'op) 'EQUAL)))
      (GPF_Gequal sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'NE))
      (GPF_Gne sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'LT))
      (GPF_Glt sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'LE))
      (GPF_Gle sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'GT))
      (GPF_Ggt sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'GE))
      (GPF_Gge sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'AND))
      (GPF_Gand sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'OR))
      (GPF_Gor sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'IMP))
      (GPF_Gimp sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'IFF))
      (GPF_Giff sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'STAR_STAR))
      (GPF_Gpower sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'STAR))
      (GPF_Gtimes sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'SLASH))
      (GPF_Gquotient sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'DIV))
      (GPF_Gdiv sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'MOD))
```

```
      (GPF_Gmod sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'PLUS))
      (GPF_Gplus sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'MINUS))
      (GPF_Gsubtract sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'IN))
      (GPF_Gin sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'ADJOIN))
      (GPF_Gadjoin sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'OMIT))
      (GPF_Gomit sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'SUB))
      (GPF_Gsub sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'UNION))
      (GPF_Gunion sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'INTERSECT))
      (GPF_Gintersect sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'DIFFERENCE))
      (GPF_Gdifference sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'COLON_GT))
      (GPF_Gcons sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'LT_COLON))
      (GPF_Grcons sV1 sV2 s0)

  (if (rule op (prodn (tag 'binary_operator 'op) 'APPEND))
      (GPF_Gappend sV1 sV2 s0)

    (mark_state_indeterminate s0))))))))))))))))))))))))))))


; ********************
;  Standard Functions
; ********************

(disable std_domain)

(constrain p_std_domain_intro (rewrite)
  (let ((s1 (p_std_domain d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_domain d) s0))
                       (determinate (std_domain d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_domain (lambda (d s0)
                    (mark_state_indeterminate s0))) ))

(defn GPF_std_domain (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_domain (result~_list sD) s0)
      r)))
```

```
(disable std_first)

(constrain p_std_first_intro (rewrite)
  (let ((s1 (p_std_first d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_first d) s0))
                       (determinate (std_first d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_first (lambda (d s0)
                   (mark_state_indeterminate s0))) ))

(defn GPF_std_first (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_first (result~_list sD) s0)
      r)))


(disable std_initial)

(constrain p_std_initial_intro (rewrite)
  (let ((s1 (p_std_initial d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_initial d) s0))
                       (determinate (std_initial d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_initial (lambda (d s0)
                     (mark_state_indeterminate s0))) ))

(defn GPF_std_initial (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_initial (result~_list sD) s0)
      r)))


(disable std_last)

(constrain p_std_last_intro (rewrite)
  (let ((s1 (p_std_last d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_last d) s0))
                       (determinate (std_last d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_last (lambda (d s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_std_last (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_last (result~_list sD) s0)
      r)))
```

```
(disable std_lower)

(constrain p_std_lower_intro (rewrite)
  (let ((s1 (p_std_lower d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_lower d) s0))
                       (determinate (std_lower d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_lower (lambda (d s0)
                   (mark_state_indeterminate s0))) ))

(defn GPF_std_lower (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_lower (result~_list sD) s0)
      r)))


(disable std_max)

(constrain p_std_max_intro (rewrite)
  (let ((s1 (p_std_max d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_max d) s0))
                       (determinate (std_max d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_max (lambda (d s0)
                 (mark_state_indeterminate s0))) ))

(defn GPF_std_max (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_max (result~_list sD) s0)
      r)))


(disable std_min)

(constrain p_std_min_intro (rewrite)
  (let ((s1 (p_std_min d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_min d) s0))
                       (determinate (std_min d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_min (lambda (d s0)
                 (mark_state_indeterminate s0))) ))

(defn GPF_std_min (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_min (result~_list sD) s0)
      r)))
```

```
(disable std_nonfirst)

(constrain p_std_nonfirst_intro (rewrite)
  (let ((s1 (p_std_nonfirst d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_nonfirst d) s0))
                       (determinate (std_nonfirst d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_nonfirst (lambda (d s0)
                      (mark_state_indeterminate s0))) ))

(defn GPF_std_nonfirst (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_nonfirst (result~_list sD) s0)
      r)))


(disable std_nonlast)

(constrain p_std_nonlast_intro (rewrite)
  (let ((s1 (p_std_nonlast d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_nonlast d) s0))
                       (determinate (std_nonlast d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_nonlast (lambda (d s0)
                     (mark_state_indeterminate s0))) ))

(defn GPF_std_nonlast (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_nonlast (result~_list sD) s0)
      r)))


(disable std_null)

(constrain p_std_null_intro (rewrite)
  (let ((s1 (p_std_null d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_null d) s0))
                       (determinate (std_null d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_null (lambda (d s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_std_null (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_null (result~_list sD) s0)
      r)))
```

```
(disable std_ord)

(constrain p_std_ord_intro (rewrite)
  (let ((s1 (p_std_ord d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_ord d) s0))
                       (determinate (std_ord d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_ord (lambda (d s0)
                 (mark_state_indeterminate s0))) ))

(defn GPF_std_ord (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_ord (result~_list sD) s0)
      r)))


(disable std_pred)

(constrain p_std_pred_intro (rewrite)
  (let ((s1 (p_std_pred d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_pred d) s0))
                       (determinate (std_pred d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_pred (lambda (d s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_std_pred (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_pred (result~_list sD) s0)
      r)))


(disable std_range)

(constrain p_std_range_intro (rewrite)
  (let ((s1 (p_std_range d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_range d) s0))
                       (determinate (std_range d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_range (lambda (d s0)
                   (mark_state_indeterminate s0))) ))

(defn GPF_std_range (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_range (result~_list sD) s0)
      r)))
```

```
(disable std_scale)

(constrain p_std_scale_intro (rewrite)
  (let ((s1 (p_std_scale d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_scale d) s0))
                       (determinate (std_scale d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_scale (lambda (d s0)
                  (mark_state_indeterminate s0)))))

(defn GPF_std_scale (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_scale (result~_list sD) s0)
      r)))


(disable std_size)

(constrain p_std_size_intro (rewrite)
  (let ((s1 (p_std_size d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_size d) s0))
                       (determinate (std_size d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_size (lambda (d s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_std_size (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_size (result~_list sD) s0)
      r)))


(disable std_succ)

(constrain p_std_succ_intro (rewrite)
  (let ((s1 (p_std_succ d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_succ d) s0))
                       (determinate (std_succ d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_succ (lambda (d s0)
                  (mark_state_indeterminate s0))) ))

(defn GPF_std_succ (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_succ (result~_list sD) s0)
      r)))
```

```
(disable std_upper)

(constrain p_std_upper_intro (rewrite)
  (let ((s1 (p_std_upper d s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (std_upper d) s0))
                       (determinate (std_upper d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_std_upper (lambda (d s0)
                   (mark_state_indeterminate s0))) ))

(defn GPF_std_upper (sD s0)
  (let ((r (state_check sD s0)))
    (if (normal_state r)
        (p_std_upper (result~_list sD) s0)
      r)))



; ***********
;  Variables
; ***********

(constrain p_apply_var_intro (rewrite)
  (let ((s1 (p_apply_var fn s0 d)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (apply_var fn (map s0) d)
                                            s0))
                       (determinate (apply_var fn (map s0) d))))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_apply_var (lambda (fn s0 d)
                   (mark_state_indeterminate s0))) ))

(defn GPF_apply_var (fn s d)
  (if (normal_state s)
      (if (state_componentp fn s)
          (let ((r (state_check d s)))
            (if (normal_state r)
                (p_apply_var fn s (result~_list d))
              r))
        (set_condition s 'routineerror))
    s))



; **************
;  Bound Values
; **************

(disable bound_values)

(constrain GPF_bound_values_intro (rewrite)
  (let ((s1 (GPF_bound_values e c s0 x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (and (sequal s1
                               (store_value 'result~ (bound_values e c x) s0))
                       (all_determinate (bound_values e c x))))
         (implies (and (determinate s1)
```

```
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (GPF_bound_values (lambda (e c s0 x)
                        (mark_state_indeterminate s0))) ))



; ***********************************
;  Space Allocation and Deallocation
; ***********************************

(constrain allocate_intro (rewrite)
  (let ((s1 (allocate k v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (store_value k v s0)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (allocate (lambda (k v s0)
                (mark_state_indeterminate s0))) ))

(constrain allocate_const_intro (rewrite)
  (let ((s1 (allocate_const k v sc s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (store_const k v sc s0)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (allocate_const (lambda (k v sc s0)
                      (mark_state_indeterminate s0))) ))

(defn GP_deallocate_locals (s)
  (deallocate_vars (local_vars s)
   (deallocate_consts (local_consts s)
    (deallocate_conds (local_conds s) s))))



; *****************
;  Name Expressions
; *****************

; A name expression is a marked object, with
;     mark = 'name_expression
;     object = (id . <evaluated selector_list>)


(defn name_exp (id ss)
  (marked 'name_expression (cons id ss)))

(defn namep (v)
  (equal (mark v) 'name_expression))

(defn ne_name (v)
  (if (namep v)
      (car (object v))
    F))

(defn ne_selectors (v)
  (if (namep v)
      (cdr (object v))
    nil))
```

```
; ******************
;  Retyping Result~
; ******************

; This is used in constant interpretation.

(defn retype_result~ (s td)
  (if (and (determinate (result~ s))
           (truep (in_type td (result~ s))))
      (store_value 'result~
                   (marked nil (typed td (value (result~ s))))
                   s)
    (set_condition s 'routineerror)))

(constrain p_retype_result~_intro (rewrite)
  (let ((s1 (p_retype_result~ s0 td)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (retype_result~ s0 td)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_retype_result~ (lambda (s0 td)
                        (mark_state_indeterminate s0))) ))

(defn GPF_retype_result~ (s td)
  (if (normal_state s)
      (p_retype_result~ s td)
    s))



; ********************************
;  Functions on Type Descriptors
; ********************************

(defn subtype_irange (t1 t2)
  (if (bounded_typep t1)
      (if (bounded_typep t2)
          (if (and (integerp (tmin t1)) (integerp (tmax t1))
                   (integerp (tmin t2)) (integerp (tmax t2)))
              (and (ileq (tmin t2) (tmin t1))
                   (ileq (tmax t1) (tmax t2)))
            F)
        T)
  (if (bounded_typep t2)
      F
    T)))

(defn subtype_rrange (t1 t2)
  (if (bounded_typep t1)
      (if (bounded_typep t2)
          (if (and (rationalp (tmin t1)) (rationalp (tmax t1))
                   (rationalp (tmin t2)) (rationalp (tmax t2)))
              (and (rleq (tmin t2) (tmin t1))
                   (rleq (tmax t1) (tmax t2)))
            F)
        T)
  (if (bounded_typep t2)
      F
    T)))

(defn subtype_size (s1 s2)
  (if (equal s1 nil)
      (equal s2 nil)
  (if (equal s2 nil)
      (numberp s1)
```

```
  (if (and (numberp s1) (numberp s2))
      (leq s1 s2)
    F))))


(do-mutual '(

(defn subtype_fields (t1 t2)
  (if (nlistp t1)
      T
    (and (subtype (cdar t1) (mapped_value t2 (caar t1)))
         (subtype_fields (cdr t1) t2)))
  ( (lessp (tree_size t1)) ))

(defn subtype (t1 t2)
  (if (and (type_descp t1) (type_descp t2)
           (equal (mode t1) (mode t2)))
      (case (mode t1)
        (integer (subtype_irange t1 t2))
        (rational (subtype_rrange t1 t2))
        (scalar (and (equal (tid t1) (tid t2))
                     (equal (sid t1) (sid t2))
                     (type_vequal (crd t1) (crd t2) (integer_desc))
                     (subtype_irange t1 t2)))
        (array (and (type_equal (selector_td t1) (selector_td t2))
                    (subtype (component_td t1) (component_td t2))))
        (record (and (set_equal (field_names t1) (field_names t2))
                     (subtype_fields (field_tds t1) (field_tds t2))))
        (mapping (and (subtype_size (max_size t1) (max_size t2))
                      (subtype (selector_td t1) (selector_td t2))
                      (subtype (component_td t1) (component_td t2))))
        (sequence (and (subtype_size (max_size t1) (max_size t2))
                       (subtype (component_td t1) (component_td t2))))
        (set (and (subtype_size (max_size t1) (max_size t2))
                  (subtype (component_td t1) (component_td t2))))
        (pending (and (equal (tid t1) (tid t2))
                      (equal (sid t1) (sid t2))))
        (otherwise F))
    F)
  ( (lessp (tree_size t1)) ))

))


(defn type_check (td s)
  (if (and (determinate (result~ s))
           (truep (in_type td (result~ s))))
      s
    (set_condition s 'routineerror)))

(constrain p_type_check_intro (rewrite)
  (let ((s1 (p_type_check td s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (type_check td s0)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_type_check (lambda (td s0)
                    (mark_state_indeterminate s0))) ))

(defn GPF_type_check (td s)
  (if (normal_state s)
      (p_type_check td s)
    s))
```

```
; *********************
;  Type Name Arguments
; *********************

(defn type_name_arg (tn sn s x)
  (store_value 'result~
               (marked 'type_descriptor
                       (type_desc (mk_identifier tn) sn nil x))
               s))

(constrain p_type_name_arg_intro (rewrite)
  (let ((s1 (p_type_name_arg tn sn s0 x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (type_name_arg tn sn s0 x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_type_name_arg (lambda (tn sn s0 x)
                       (mark_state_indeterminate s0))) ))

(defn GPF_type_name_arg (tn sn s x)
  (if (normal_state s)
      (p_type_name_arg tn sn s x)
    s))



; **********************
;  Specification Values
; **********************

(defn set_entry (e c s n x)
  (let ((v (GF e c (map s) n x)))
    (if (boolean_typep (type v))
        (store_value 'entry v s)
      (store_value 'entry
                   (marked (entry_not_boolean_error e c)
                           (default_value (boolean_desc)))
                   s))))

(constrain p_set_entry_intro (rewrite)
  (let ((s1 (p_set_entry e c s0 n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (set_entry e c s0 n x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_set_entry (lambda (e c s0 n x)
                   (mark_state_indeterminate s0))) ))

(defn GP_set_entry (e c s n x)
  (if (normal_state s)
      (p_set_entry e c s n x)
    s))

(defn exit_labels_ok (cs s)
  (and (setp cs)
       (all_conditionsp cs (store_cond 'normal 'formal s))))

(defn set_exit (e c s n x)
  ; e is an opt_exit_specification
  (if (exit_labels_ok (exit_labels e) s)
      (let ((v (GF (postc e (cond~ s)) c (map s) n x)))
```

```
        (if (boolean_typep (type v))
            (store_value 'exit v s)
          (store_value 'exit
                       (marked (exit_not_boolean_error e c)
                               (default_value (boolean_desc)))
                       s)))
    (store_value 'exit
                 (marked (exit_label_error e c)
                         (default_value (boolean_desc)))
                 s)))

(constrain p_set_exit_intro (rewrite)
  (let ((s1 (p_set_exit e c s0 n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (set_exit e c s0 n x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_set_exit (lambda (e c s0 n x)
                  (mark_state_indeterminate s0))) ))

(defn GP_set_exit (e c s n x)
  ; e is an opt_exit_specification
  (if (normal_state s)
      (p_set_exit e c s n x)
    s))

(defn update_keep (s c n x)
  (store_value 'keep
               (Gand (keep s)
                     (GF (keep~ s) c (map s) n x))
               s))

(constrain p_update_keep_intro (rewrite)
  (let ((s1 (p_update_keep s0 c n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (update_keep s0 c n x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_update_keep (lambda (s0 c n x)
                      (mark_state_indeterminate s0))) ))

(defn GP_update_keep (s c n x)
  (if (normal_state s)
      (p_update_keep s c n x)
    s))

(defn GP_set_keep (k s c n x)
  (if (normal_state s)
      (GP_update_keep (allocate 'keep~ k s) c n x)
    s))

(defn update_assert (e c s n x)
  (store_value 'assert
               (Gand (assert s)
                     (GF e c (map s) n x))
               s))

(constrain p_update_assert_intro (rewrite)
  (let ((s1 (p_update_assert e c s0 n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
```

```
                  (sequal s1
                          (update_assert e c s0 n x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_update_assert (lambda (e c s0 n x)
                       (mark_state_indeterminate s0))) ))

(defn GP_update_assert (e c s n x)
  (if (normal_state s)
      (p_update_assert e c s n x)
    s))

(defn record_assert (v s)
  (store_value 'assert
               (Gand (assert s) v)
               s))

(constrain p_record_assert_intro (rewrite)
  (let ((s1 (p_record_assert v s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (record_assert v s0)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_record_assert (lambda (v s0)
                       (mark_state_indeterminate s0))) ))

(defn GP_record_assert (v s)
  (if (normal_state s)
      (p_record_assert v s)
    s))



; ************
;  Assignment
; ************

(defn mapping_selectionp (ss td)
  (if (nlistp ss)
      F
    (case (mode td)
      (array (mapping_selectionp (cdr ss) (component_td td)))
      (record (mapping_selectionp (cdr ss) (field_td (value (car ss)) td)))
      (mapping T)
      (sequence (mapping_selectionp (cdr ss) (component_td td)))
      (otherwise F))))

(defn mapping_element_lhsp (n s)
  ; n is a name expression, a marked object with mark 'name_expression and
  ;   object (<id> . <evaluated selector list>).  The <evaluated selector list>
  ;   is a list of marked typed values.
  ; s is the state
  (if (namep n)
      (mapping_selectionp (ne_selectors n) (type_of (ne_name n) s))
    F))

(defn Gassign0 (ne v s)
  ; ne is a name expression, a marked object with mark 'name_expression and
  ;    object (<id> . <evaluated selector list>). The <evaluated selector list>
  ;    is a list of marked typed values.
  ; v is the new (marked typed) value
  ; s is the state
  (if (and (namep ne)
```

```
           (variablep (ne_name ne) s)
           (not (mapping_element_lhsp ne s)))
      (let ((id (ne_name ne))
            (ss (ne_selectors ne)))
        (if (state_componentp id s)
            (let ((td (type_of id s))
                  (v2 (put_op (state_component id s) ss v)))
              (if (and (determinate v2) (truep (in_type td v2)))
                  (store_value id
                               (marked nil (typed td (value v2)))
                               s)
                (set_condition s 'routineerror)))
          (set_condition s 'routineerror)))
    (set_condition s 'routineerror)))

(defn Gassign (ne v s c n x)
  ; ne is a name expression, a marked object with mark 'name_expression and
  ;    object (<id> . <evaluated selector list>). The <evaluated selector list>
  ;    is a list of marked typed values.
  ; v is the new (marked typed) value
  ; s is the state
  ; c, n, and x are the usual
  (GP_update_keep (Gassign0 ne v s) c n x))

(constrain p_assign_intro (rewrite)
  (let ((s1 (p_assign ne v s0 c n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (Gassign ne v s0 c n x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_assign (lambda (ne v s0 c n x)
                (mark_state_indeterminate s0))) ))

(defn GP_assign (sNE sV s0 c n x)
  (let ((r (state_check (list sNE sV) s0)))
    (if (normal_state r)
        (p_assign (result~ sNE) (result~ sV) s0 c n x)
      r)))



; ***************
;  New Statement
; ***************

(defn Gnew0 (dc v ne s)
  ; dc is the parse tree for the new_dynamic_variable_component
  ; v is the new (marked typed) value
  ; ne the name from the new_dynamic_variable_component
  (let ((id (ne_name ne))
        (ss (ne_selectors ne)))

    (if (rule dc (prodn (tag 'new_dynamic_variable_component 'dc)
                        (list 'INTO (tag 'name_expression 'ne))))
        (Gassign0 (name_exp id (rcdr ss))
                  (Gmap_insert (apply_var id (map s) (rcdr ss))
                               (rcar ss)
                               v)
                  s)

    (if (rule dc (prodn (tag 'new_dynamic_variable_component 'dc)
                        (list 'INTO 'SET (tag 'name_expression 'ne))))
        (Gassign0 ne (Gadjoin (apply_var id (map s) ss) v) s)

    (if (rule dc (prodn (tag 'new_dynamic_variable_component 'dc)
```

```
                        (list 'BEFORE (tag 'name_expression 'ne))))
        (Gassign0 (name_exp id (rcdr ss))
                  (Gseq_insert_before (apply_var id (map s) (rcdr ss))
                                      (rcar ss)
                                      v)
                  s)

    (if (rule dc (prodn (tag 'new_dynamic_variable_component 'dc)
                        (list 'BEFORE 'SEQ (tag 'name_expression 'ne))))
        (Gassign0 ne (Gcons v (apply_var id (map s) ss)) s)

    (if (rule dc (prodn (tag 'new_dynamic_variable_component 'dc)
                        (list 'BEHIND (tag 'name_expression 'ne))))
        (Gassign0 (name_exp id (rcdr ss))
                  (Gseq_insert_behind (apply_var id (map s) (rcdr ss))
                                      (rcar ss)
                                      v)
                  s)

  (if (rule dc (prodn (tag 'new_dynamic_variable_component 'dc)
                      (list 'BEHIND 'SEQ (tag 'name_expression 'ne))))
      (Gassign0 ne (Grcons (apply_var id (map s) ss) v) s)

    (mark_state_indeterminate s)))))))))

(defn Gnew (dc v ne c s n x)
  ; dc is the parse tree for the new_dynamic_variable_component
  ; v is the new (marked typed) value
  ; ne the name from the new_dynamic_variable_component
  ; s is the state
  ; c, n, and x are the usual
  (GP_update_keep (Gnew0 dc v ne s) c n x))

(constrain p_new_intro (rewrite)
  (let ((s1 (p_new dc v ne c s0 n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (Gnew dc v ne c s0 n x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_new (lambda (dc v ne c s0 n x)
             (mark_state_indeterminate s0))) ))

(defn GP_new (dc sV sNE c s0 n x)
  ; dc is the parse tree for the new_dynamic_variable_component
  ; sV is the state whose result~ component is the new value
  ; sNE is the state whose result~ component is the name from the
  ;     new_dynamic_variable_component
  (let ((r (state_check (list sV sNE) s0)))
    (if (normal_state r)
        (p_new dc (result~ sV) (result~ sNE) c s0 n x)
      r)))



; *****************
;  Remove Statement
; *****************

(defn Gremove0 (v ne s)
  ; v is either nil or the (marked typed) value to be removed from a set
  ; ne is the name of either the set from which v is to be removed or the
  ;    element that is to be removed
  (let ((id (ne_name ne))
        (ss (ne_selectors ne)))
    (if (equal v nil)
```

```
        (if (mapping_descp (type (apply_var id (map s) (rcdr ss))))
            (Gassign0 (name_exp id (rcdr ss))
                      (Gmapomit (apply_var id (map s) (rcdr ss))
                                (rcar ss))
                      s)
          (Gassign0 (name_exp id (rcdr ss))
                    (Gseqomit (apply_var id (map s) (rcdr ss))
                              (rcar ss))
                    s))
      (Gassign0 ne (Gomit (apply_var id (map s) ss) v) s))))

(defn Gremove (v ne c s n x)
  ; v is either nil or the (marked typed) value to be removed from a set
  ; ne is the name of either the set from which v is to be removed or the
  ;    element that is to be removed
  ; s is the state
  ; c, n, and x are the usual
  (GP_update_keep (Gremove0 v ne s) c n x))

(constrain p_remove_intro (rewrite)
  (let ((s1 (p_remove v ne c s0 n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (Gremove v ne c s0 n x)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_remove (lambda (v ne c s0 n x)
                (mark_state_indeterminate s0))) ))

(defn GP_remove (sV sNE c s0 n x)
  ; sV is either nil or the state whose result~ component is the value to be
  ;     removed from a set
  ; sNE is the state whose result~ component is the name of either the set
  ;     from which (result~ sV) is to be removed or the element that is to
  ;     be removed
  (let ((r (if (equal sV nil)
               (state_check (list sNE) s0)
             (state_check (list sV sNE) s0))))
    (if (normal_state r)
        (p_remove (if (equal sV nil) nil (result~ sV))
                  (result~ sNE) c s0 n x)
      r)))



; ****************
;  Move Statement
; ****************

(defn stored_value (n s)
  (apply_var (ne_name n) (map s) (ne_selectors n)))

(defn Gmove_assign (cd v ne s)

  (if (rule cd (prodn (tag 'component_destination 'd)
                      (tag 'new_dynamic_variable_component 'dc)))
      (Gnew0 (subtree cd 'new_dynamic_variable_component)
             v ne s)

  (if (rule cd (prodn (tag 'component_destination 'd)
                      (list 'TO (tag 'name_expression 'ne))))
      (if (sequence_descp (type (stored_value
                                 (name_exp (ne_name ne)
                                           (rcdr (ne_selectors ne)))
                                 s)))
          (Gassign0 ne v s)
```

```
        (set_condition s 'routineerror))

    (mark_state_indeterminate s))))

(defn same_selectors (s1 s2)
  (if (nlistp s1)
      (nlistp s2)
  (if (nlistp s2)
      F
    (and (or (equal (car s1) (car s2)) ; for record field names
             (Gtruep (Gequal (car s1) (car s2))))
         (same_selectors (cdr s1) (cdr s2))))))

(defn same_names (n1 n2)
  (and (namep n1) (namep n2)
       (equal (ne_name n1) (ne_name n2))
       (same_selectors (ne_selectors n1) (ne_selectors n2))))

(defn remove_dynamic_name (Rv Rne)
  (if (equal Rv nil)
      (name_exp (ne_name Rne) (rcdr (ne_selectors Rne)))
    Rne))

(defn assign_dynamic_name (cd Nne)

  (if (rule cd (prodn (tag 'component_destination 'd)
                      (tag 'new_dynamic_variable_component 'dc)))
      (assign_dynamic_name (subtree cd 'new_dynamic_variable_component) Nne)

  (if (or (rule cd (prodn (tag 'new_dynamic_variable_component 'dc)
                          (list 'INTO (tag 'name_expression 'ne))))
          (rule cd (prodn (tag 'new_dynamic_variable_component 'dc)
                          (list 'BEFORE (tag 'name_expression 'ne))))
          (rule cd (prodn (tag 'new_dynamic_variable_component 'dc)
                          (list 'BEHIND (tag 'name_expression 'ne))))
          (rule cd (prodn (tag 'component_destination 'd)
                          (list 'TO (tag 'name_expression 'ne)))))
      (name_exp (ne_name Nne) (rcdr (ne_selectors Nne)))

    Nne))

  ( (lessp (tree_size cd)) ))


(defn Gmove (Rv Rne cd Nne c s n x)
  ; Rv is either nil or the (marked typed) value to be moved from a set
  ; Rne is the name of either the set from which Rv is to be moved or
  ;     the element that is to be moved
  ; cd is the parse tree for the component_destination
  ; Nne is the name from the component_destination
  (if (same_names (remove_dynamic_name Rv Rne)
                  (assign_dynamic_name cd Nne))
      (set_condition s 'routineerror)
    (let ((r1 (Gmove_assign cd (if (equal Rv nil) (stored_value Rne s) Rv)
                            Nne s)))
      (if (normal_state r1)
          (let ((r2 (Gremove0 Rv Rne s)))
            (if (normal_state r2)
                (GP_update_keep r2 c n x)
              (marked (mark r2)
                      (map (set_condition s (cond~ r2))))))
        r1))))

(constrain p_move_intro (rewrite)
  (let ((s1 (p_move Rv Rne cd Nne c s0 n x)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (Gmove Rv Rne cd Nne c s0 n x)))
```

```
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_move (lambda (Rv Rne cd Nne c s0 n x)
              (mark_state_indeterminate s0))) ))

(defn GP_move (sRV sRNE cd sNNE c s n x)
  ; sRV is either nil or the state whose result~ component is the value to be
  ;     moved from a set
  ; sRNE is the state whose result~ component is the name of either the set
  ;      from which (result~ sRV) is to be moved or the element that is to
  ;      be moved
  ; cd is the parse tree for the component_destination
  ; sNNE is the state whose result~ component is the name of the move
  ;      destination
  (let ((r (if (equal sRV nil)
               (state_check (list sRNE sNNE) s)
             (state_check (list sRV sRNE sNNE) s))))
    (if (normal_state r)
        (p_move (if (equal sRV nil) nil (result~ sRV))
                (result~ sRNE) cd (result~ sNNE) c s n x)
      r)))



; *****************
;  Procedure Calls
; *****************

; ======================================
;  Argument Checking for Procedure Calls
; ======================================

; -----------
;  Formals OK
; -----------

(defn pformals_ok (fs)
  (if (nlistp fs)
      T
  (if (reserved_idp (car fs))
      F
  (if (member (car fs) (cdr fs))
      F
    (pformals_ok (cdr fs))))))


; ----------------
;  Aliasing Checks
; ----------------

(defn selectors_aliasedp (s1 s2)
  (if (or (nlistp s1) (nlistp s2))
      T
    (and (or (equal (car s1) (car s2)) ; for record field names
             (Gtruep (Gequal (car s1) (car s2))))
         (selectors_aliasedp (cdr s1) (cdr s2)))))

(defn harmful_aliasp (a1 a2 f1 f2)
  ; Check to see if actual parameters a1 and a2 are harmful aliases.
  ; f1 and f2 are the corresponding formals
  (if (or (equal (access f1) 'var)
          (equal (access f2) 'var))
      (if (and (namep a1) (namep a2))
          (and (equal (ne_name a1) (ne_name a2))
               (selectors_aliasedp (ne_selectors a1) (ne_selectors a2)))
        F)
    F))
```

```
(defn harmfully_aliasedp (ap as fp fs)
  ; Check that actual parameter ap is not harmfully aliased in as.
  ; as is the list of actual parameters
  ; fp is the formal parameter correcsponding to ap
  ; fs is the list of formal parameters corresponding to as
  ; entry (equal (length as) (length fs))
  ;       & all variable parameters have var actuals
  (if (nlistp as)
      F
    (or (harmful_aliasp ap (car as) fp (car fs))
        (harmfully_aliasedp ap (cdr as) fp (cdr fs)))))

(defn no_harmful_aliasing (as fs)
  ; as is the list of actual parameters
  ; fs is the list of formal parameters
  ; entry   (equal (length as) (length fs))
  ;       & all variable parameters have var actuals
  (if (nlistp as)
      T
  (if (namep (car as))
      (and (not (harmfully_aliasedp (car as) (cdr as) (car fs) (cdr fs)))
           (no_harmful_aliasing (cdr as) (cdr fs)))
    (no_harmful_aliasing (cdr as) (cdr fs)))))


; -----------------------
;  Type and Access Checks
; -----------------------

(defn one_parg_check (fp ap fsn s x)
  (let ((ft (type_desc (formal_type fp) fsn nil x))
        (av (if (namep ap)
                (apply_var (ne_name ap) (map s) (ne_selectors ap))
              ap)))
    (if (indeterminate av)
        (set_condition s 'routineerror)
    (if (equal (access fp) 'var)
        (if (and (namep ap)
                 (variablep (ne_name ap) s)
                 (not (mapping_element_lhsp ap s)))
            (if (and (subtype ft (type av))
                     (truep (in_type ft av)))
                s
              (set_condition s 'routineerror))
          (set_condition s 'routineerror))
    (if (truep (in_type ft av))
        s
      (set_condition s 'routineerror))))))

(disable set_condition)
(disable one_parg_check)
(disable normal_state)

(defn parg_check2 (fs as fsn s x)
  (if (nlistp as)
      (if (nlistp fs)
          s
        (set_condition s 'routineerror))
  (if (nlistp fs)
      (set_condition s 'routineerror)
    (let ((s2 (one_parg_check (car fs) (car as) fsn s x)))
      (if (normal_state s2)
          (parg_check2 (cdr fs) (cdr as) fsn s x)
        s2)))))


; -------------------------------------------
;  The Full Argument Check for Procedure Calls
; -------------------------------------------
```

```
(defn cond_arg_check (fcs acs s)
  (if (equal (length fcs) (length acs))
      (if (all_conditionsp acs s)
          s
        (mark_state_indeterminate s))
    (set_condition s 'routineerror)))

(defn parg_check (u usn ads acs s x)
  ; u is the unit declaration for the called procedure
  ; usn is the name of the scope where u is declared
  ; ads is the actual data parameters
  ; acs is the actual condition parameters
  ; s is the state from which u is called
  ; x is the Gypsy program_description
  (let ((fds (formal_dargs u))
        (fcs (formal_cargs u)))
    (case (kind u)
      (function (if (and (pformals_ok
                              (append (cons 'result (dparam_name_list fds))
                                      (append fcs
                                              '(routineerror spaceerror))))
                         (equal (farg_check fds ads usn x) nil))
                    (cond_arg_check fcs acs s)
                  (set_condition s 'routineerror)))
      (procedure (if (pformals_ok (append
                                    (dparam_name_list fds)
                                    (append fcs
                                            '(routineerror spaceerror))))
                     (let ((s2 (parg_check2 fds ads usn s x)))
                       (if (normal_state s2)
                           (if (no_harmful_aliasing ads fds)
                               (cond_arg_check fcs acs s)
                             (set_condition s 'routineerror))
                         s2))
                   (set_condition s 'routineerror)))
      (otherwise (set_condition s 'routineerror)))))



; ==============================
;  New State for Procedure Call
; ==============================

(defn padd_darg (fp ap fsn Sa s x)
  ; fp is the formal
  ; ap is the actual
  ; fsn is the name of f's scope
  ; Sa is a's state
  ; s is the new state being built
  ; x is the Gypsy program_description
  ; entry (truep (in_type ft av))
  (let ((ft (type_desc (formal_type fp) fsn nil x))
        (av (if (namep ap)
                (apply_var (ne_name ap) (map Sa) (ne_selectors ap))
              ap)))
  (let ((fv (marked nil (typed ft (value av)))))
    (if (equal (access fp) 'var)
        (store_const (mk_entry_name (dparam_name fp)) fv 'formal
                     (store_var (dparam_name fp) fv 'formal s))
      (store_const (dparam_name fp) fv 'formal s)))))

(defn pbind_dargs (fs as fsn s x)
  (if (nlistp as)
      (default_state)
    (padd_darg (rcar fs) (rcar as) fsn s
               (pbind_dargs (rcdr fs) (rcdr as) fsn s x)
               x)))

(defn padd_result (s ftype)
```

```
  (let ((v (std_initial (list (marked 'type_descriptor ftype)))))
    (if (determinate v)
        (store_const (mk_entry_name 'result) v 'formal
                     (store_var 'result v 'formal s))
      (set_condition s 'routineerror))))

(defn call_state (u usn ads acs s x)
  (let ((r1 (parg_check u usn ads acs s x)))
    (if (normal_state r1)
        (let ((r2 (note_conds (formal_cargs u) 'formal
                              (pbind_dargs (formal_dargs u) ads usn s x))))
          (if (equal (kind u) 'function)
              (let ((ftype (type_desc (result_type u) usn nil x)))
                (padd_result r2 ftype))
            r2))
      r1)))

(disable call_state)
(disable determinate)
(enable sequal)

(constrain p_call_state_intro (rewrite)
  (implies (and (determinate (p_call_state u usn ads acs s0 x))
                (not (sequal (p_call_state u usn ads acs s0 x)
                             (call_state u usn ads acs s0 x))))
           (member (cond~ (p_call_state u usn ads acs s0 x))
                   '(routineerror spaceerror)))
  ( (p_call_state (lambda (u usn ads acs s0 x)
                    (call_state u usn ads acs s0 x))) ))

(enable determinate)
(disable sequal)

(defn GP_call_state (u usn Sads acs s x)
  (let ((r (state_check Sads s)))
    (if (normal_state r)
        (p_call_state u usn (result~_list Sads) acs s x)
      r)))


; ==============================
;  State Update for Call Effects
; ==============================

(defn map_cond_effects (fc fcs acs s)
  ; entry (equal (length fcs) (length acs))
  (if (equal fc 'normal)
      s
  (if (member fc '(routineerror spaceerror))
      (set_condition s fc)
  (if (and (listp fcs) (listp acs))
      (if (equal fc (car fcs))
          (set_condition s (car acs))
        (map_cond_effects fc (cdr fcs) (cdr acs) s))
    (mark_state_indeterminate s)))))

(defn map_var_effects (fs as s2 s1)
  ; fs is the formal data parameters
  ; as is the actual data parameters
  ; s2 is the state after the called procedure finished running
  ; s1 is the state being updated for call effects
  (if (nlistp fs)
      s1
  (if (equal (access (car fs)) 'var)
      ; (car as) is a name expression because all the actual/formal arg
      ; checks have been done.  For the same reason, Gassign0 below will
      ; take place normally.
      (map_var_effects (cdr fs) (cdr as) s2
                       (Gassign0 (car as)
```

```
                                 (state_component (dparam_name (car fs)) s2)
                                 s1))
    (map_var_effects (cdr fs) (cdr as) s2 s1))))

(defn map_call_effects (s2 u ads acs c s1 n x)
  ; s2 is the state after a called procedure finished running
  ; u is the unit declaration of the called procedure
  ; ads is the actual data parameters
  ; acs is the actual condition parameters
  ; c is the name of the scope from which the call took place
  ; s1 is the state before the procedure was called, a normal state
  ; n and x are as usual

  (if (indeterminate s2)
      (mark_state_indeterminate s1)
  (if (equal (kind u) 'function)
      (if (condition_normal s2)
          (allocate 'result~ (state_component 'result s2) s1)
        (map_cond_effects (cond~ s2) (formal_cargs u) acs s1))
    ; (equal (kind u) 'procedure)
    (GP_update_keep
       (map_cond_effects (cond~ s2) (formal_cargs u) acs
                         (map_var_effects (formal_dargs u) ads s2 s1))
       c n x))))

(enable sequal)

(constrain GP_map_call_effects_intro (rewrite)
  ; Implementation can't signal any conditions not covered by
  ; map_call_effects.
  (sequal (GP_map_call_effects s2 u ads acs c s1 n x)
          (map_call_effects s2 u ads acs c s1 n x))
  ( (GP_map_call_effects (lambda (s2 u ads acs c s1 n x)
                          (map_call_effects s2 u ads acs c s1 n x))) ))

(disable sequal)


; =============
;  Local Names
; =============

(defn GP_new_namep (id s)
  (and (not (in_map (map s) id))
       (not (in_map (cond+ s) id))
       (not (reserved_idp id))))


; =================
;  Local Variables
; =================

(defn bind_local (a id td iv s)
  ; a is 'var or 'const
  ; id is a local name
  ; td is the type descriptor for id
  ; iv is nil or the (marked typed) value from evaluating the default initial
  ;    value expression
  ; s is the state
  (if (GP_new_namep id s)
      (let ((v (if (equal iv nil)
                   (std_initial (list (marked 'type_descriptor td)))
                 iv)))
        (if (and (determinate v) (truep (in_type td v)))
            (if (equal a 'var)
                (store_var id (marked nil (typed td (value v))) 'local s)
              (store_const id (marked nil (typed td (value v))) 'local s))
          (set_condition s 'routineerror)))
    (set_condition s 'routineerror)))
```

```
(constrain p_bind_local_intro (rewrite)
  (let ((s1 (p_bind_local a id td iv s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (bind_local a id td iv s0)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
  ( (p_bind_local (lambda (a id td iv s0)
                    (mark_state_indeterminate s0))) ))

(defn GP_bind_local (a id td iv s)
  ; a is 'var or 'const
  ; id is a local name
  ; td is the type descriptor for id
  ; iv is nil or the state resulting from evaluating the default initial value
  ;    expression
  ; s is the state
  (let ((r (if (equal iv nil) s (state_check (list iv) s))))
    (if (normal_state r)
        (p_bind_local a id td (if (equal iv nil) iv (result~ iv)) s)
      r)))

(defn GP_bind_locals (a ids td iv s)
  ; a is 'var or 'const
  ; ids is a list of local names
  ; td is the type descriptor for the ids
  ; iv is nil or the state resulting from evaluating the default initial value
  ;    expression
  ; s is the state
  (if (nlistp ids)
      s
    (GP_bind_locals a (cdr ids) td iv
                    (GP_bind_local a (car ids) td iv s))))

(defn GP_local_conds (ids s)
  (if (or (nlistp ids) (not (normal_state s)))
      s
  (if (GP_new_namep (car ids) s)
      (GP_local_conds (cdr ids)
                      (store_cond (car ids) 'local s))
    (set_condition s 'routineerror))))



; ****************
;  Case Statement
; ****************

(defn case_label_check (k cs s)
  (let ((td (base_type (type k))))
    (if (and (non_rational_simple_typep td)
             (equal (arg_check cs (ncopies (length cs) td)) nil)
             (vsetp (values cs) td))
        s
      (set_condition s 'routineerror))))

(constrain p_case_label_check_intro (rewrite)
  (let ((s1 (p_case_label_check k cs s0)))
    (and (implies (and (determinate s1)
                       (equal (cond~ s1) 'normal))
                  (sequal s1
                          (case_label_check k cs s0)))
         (implies (and (determinate s1)
                       (not (equal (cond~ s1) 'normal)))
                  (and (implementation_constrained s1 s0)
                       (member (cond~ s1) '(routineerror spaceerror))))))
```

```
  ( (p_case_label_check (lambda (k cs s0)
                          (mark_state_indeterminate s0))) ))

(defn GP_case_label_check (sK sCS s0)
  (let ((r (state_check (cons sK sCS) s0)))
    (if (normal_state r)
        (p_case_label_check (result~ sK) (result~_list sCS) s0)
      r)))

(defn condition_labels_ok (cs s)
  (and (setp cs)
       (all_conditionsp cs s)))



; **********************
;  The Meta-Function GP
; **********************

(disable *1*boolean_desc)
(disable GPF_Gand)
(disable GPF_Gchar)
(disable GPF_Gin)
(disable GPF_Gmap_insert)
(disable GPF_Gmapomit)
(disable GPF_Gor)
(disable GPF_Grange_elements)
(disable GPF_Gseq_insert_before)
(disable GPF_Gseq_insert_behind)
(disable GPF_Gseqomit)
(disable GPF_Gset)
(disable GPF_Gset_or_seq)
(disable GPF_Gstring_seq)
(disable GPF_apply_binary_op)
(disable GPF_apply_unary_op)
(disable GPF_apply_var)
(disable GPF_bound_values)
(disable GPF_false)
(disable GPF_minteger)
(disable GPF_put_op)
(disable GPF_record_get)
(disable GPF_retype_result~)
(disable GPF_select_op)
(disable GPF_std_domain)
(disable GPF_std_first)
(disable GPF_std_initial)
(disable GPF_std_last)
(disable GPF_std_lower)
(disable GPF_std_max)
(disable GPF_std_min)
(disable GPF_std_nonfirst)
(disable GPF_std_nonlast)
(disable GPF_std_null)
(disable GPF_std_ord)
(disable GPF_std_pred)
(disable GPF_std_range)
(disable GPF_std_scale)
(disable GPF_std_size)
(disable GPF_std_succ)
(disable GPF_std_upper)
(disable GPF_subsequence_get)
(disable GPF_true)
(disable GPF_type_check)
(disable GPF_type_name_arg)
(disable GP_assign)
(disable GP_bind_locals)
(disable GP_call_state)
(disable GP_case_label_check)
(disable GP_deallocate_locals)
```

```
(disable GP_local_conds)
(disable GP_map_call_effects)
(disable GP_move)
(disable GP_new)
(disable GP_new_namep)
(disable GP_record_assert)
(disable GP_remove)
(disable GP_set_entry)
(disable GP_set_exit)
(disable GP_set_keep)
(disable GP_update_assert)
(disable Gtruep)
(disable access)
(disable actual_cargs)
(disable actual_dargs)
(disable allocate)
(disable allocate_const)
(disable arg_list)
(disable base_type)
(disable boolean_desc)
(disable bound_id)
(disable case_labels)
(disable cdr_quantified_exp)
(disable character_valuep)
(disable condition_labels_ok)
(disable condition_non_normal)
(disable condition_normal)
(disable conditionp)
(disable cond~)
(disable constant_body)
(disable digit_listp)
(disable each_clausep)
(disable entry_name)
(disable entry_valuep)
(disable exit_spec)
(disable expression_from_spec)
(disable fn_call_formp)
(disable formal_cargs)
(disable formal_dargs)
(disable gname)
(disable handler)
(disable handler_labels)
(disable id_list)
(disable identifierp)
(disable if_else_exp)
(disable if_statement_else_part)
(disable indeterminate)
(disable internal_initial_value_exp)
(disable keep_spec)
(disable kind)
(disable length)
(disable map)
(disable map_cond_effects)
(disable mark_state_indeterminate)
(disable mk_name_expression)
(disable mk_signal_stmt)
(disable mk_unary_operator)
(disable name_exp)
(disable new_name_arg)
(disable normal_state)
(disable object_name)
(disable prec)
(disable procedure_body)
(disable rcar)
(disable rcdr)
(disable ref)
(disable remove_exp_arg)
(disable remove_name_arg)
(disable reset_leave_to_normal)
```

```
(disable result_type)
(disable result~)
(disable result~_list)
(disable set_condition)
(disable state_componentp)
(disable string_valuep)
(disable type)
(disable type_desc)
(disable type_name_expp)


(do-mutual '(

; ****************************************************************************
;
;                       THE EXPRESSION INTERPRETER
;
; ****************************************************************************

; ***************************
;  Set/Sequence Constructors
; ***************************

;; This returns a list of states rather than a single state.

(defn GPF_element_list (e c s n x)

  (if (not (normal_state s))
      (list s)

  (if (zerop (fix n))
      (list (mark_state_indeterminate s))

  (if (rule e (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GPF_element_list (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'value_list 'v)))
      (GPF_element_list (subtree e 'value_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'range_limits 'r)))
      (GPF_element_list (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (GPF_Grange_elements (GPF (subtree_i e 'expression 1) c s (sub1 n) x)
                           (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
                           s)

  (if (rule e (prodn (tag 'value_list 'v)
                     (tag 'expression 'e)))
      (rcons nil (GPF (subtree e 'expression) c s (sub1 n) x))

  (if (rule e (prodn (tag 'value_list 'v)
                     (list (tag 'value_list 'v2) 'COMMA
                           (tag 'expression 'e))))
      (rcons (GPF_element_list (subtree e 'value_list) c s (sub1 n) x)
             (GPF (subtree e 'expression) c s (sub1 n) x))

    (list (mark_state_indeterminate s)))))))))

  ( (lessp (count n)) ))


; Unlike most of the other functions here, this doesn't store the value in
; the state.
```

```
(defn GPF_element_type (e c s n x)

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule e (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2) 'CLOSE_PAREN)))
      (GPF_element_type (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'value_list 'v)))
      (GPF_element_type (subtree e 'value_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'element_list 'e)
                     (tag 'range_limits 'r)))
      (GPF_element_type (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (base_type (type (result~ (GPF (subtree_i e 'expression 1)
                                     c s (sub1 n) x))))

  (if (rule e (prodn (tag 'value_list 'v)
                     (tag 'expression 'e)))
      (base_type (type (result~ (GPF (subtree e 'expression) c s (sub1 n) x))))

  (if (rule e (prodn (tag 'value_list 'v)
                     (list (tag 'value_list 'v2) 'COMMA
                           (tag 'expression 'e))))
      (GPF_element_type (subtree e 'value_list) c s (sub1 n) x)

    nil)))))))

  ( (lessp (count n)) ))



; ************************
;  Quantified expressions
; ************************

(defn GPF_all (id vs e c s n x)
  (if (nlistp vs)
      (GPF_true s)
  (if (GP_new_namep id s)
      (if (zerop n)
          (mark_state_indeterminate s)
        (GPF_Gand (GPF_all id (rcdr vs) e c s (sub1 n) x)
                  (GPF e c (allocate_const id (rcar vs) 'local s) (sub1 n) x)
                  s))
    (set_condition s 'routineerror)))
  ( (lessp (count n)) ))


(defn GPF_some (id vs e c s n x)
  (if (nlistp vs)
      (GPF_false s)
  (if (GP_new_namep id s)
      (if (zerop n)
          (mark_state_indeterminate s)
        (GPF_Gor (GPF_some id (rcdr vs) e c s (sub1 n) x)
                 (GPF e c (allocate_const id (rcar vs) 'local s) (sub1 n) x)
                 s))
    (set_condition s 'routineerror)))
  ( (lessp (count n)) ))


; ********************
```

```
;  Value Modifications
; ********************

(defn GPF_each (id vs sBV e c s n x)
  ; e is the <component modification>
  ; sBV is the state whose result~ component is the value being modified
  (if (not (normal_state s))
      s
  (if (nlistp vs)
      sBV
  (if (GP_new_namep id s)
      (if (zerop n)
          (mark_state_indeterminate s)
        (GPF_each id (cdr vs)
                  (GPF_modifiers sBV e c (allocate_const id (car vs) 'local s)
                                 (sub1 n) x)
                  e c s (sub1 n) x))
    (set_condition s 'routineerror))))
  ( (lessp (count n)) ))


(defn GPF_adp (e c s n x)

  (if (not (normal_state s))
      (list s)

  (if (zerop (fix n))
      (list (mark_state_indeterminate s))

  (if (rule e (prodn (tag 'arg_list 'as)
                     (list 'OPEN_PAREN (tag 'value_list 'vs)
                           'CLOSE_PAREN)))
      (GPF_adp (subtree e 'value_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'value_list 'vs)
                     (tag 'expression 'e)))
      (rcons nil (GPF (subtree e 'expression) c s (sub1 n) x))

  (if (rule e (prodn (tag 'value_list 'vs)
                     (list (tag 'value_list 'vs2)
                           'COMMA (tag 'expression 'e))))
      (rcons (GPF_adp (subtree e 'value_list) c s (sub1 n) x)
             (GPF (subtree e 'expression) c s (sub1 n) x))

    (list (mark_state_indeterminate s)))))))

  ( (lessp (count n)) ))

(defn GPF_selectors (e c s n x)

  (if (not (normal_state s))
      (list s)

  (if (zerop (fix n))
      (list (mark_state_indeterminate s))

  (if (rule e (prodn (tag 'selector_list 's)
                     (tag 'component_selectors 's2)))
      (GPF_selectors (subtree e 'component_selectors) c s (sub1 n) x)

  (if (rule e (prodn (tag 'selector_list 's)
                     (list (tag 'selector_list 's2)
                           (tag 'component_selectors 's3))))
      (append (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x)
              (GPF_selectors (subtree e 'component_selectors) c s (sub1 n) x))

  (if (rule e (prodn (tag 'component_selectors 's)
                     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (list (allocate 'result~
```

```
                      (marked 'field_name (gname (subtree e 'IDENTIFIER)))
                      s))

  (if (rule e (prodn (tag 'component_selectors 's)
                     (tag 'arg_list 'd)))
      (GPF_adp (subtree e 'arg_list) c s (sub1 n) x)

  (if (rule e (prodn (tag 'arg_list 'as)
                     (list 'OPEN_PAREN (tag 'value_list 'vs)
                           'CLOSE_PAREN)))
      (GPF_adp (subtree e 'value_list) c s (sub1 n) x)

    (list (mark_state_indeterminate s)))))))))

  ( (lessp (count n)) ))

(defn GPF_modifiers (sBV e c s n x)
  ; e is the <value modifiers>
  ; sBV is the state whose result~ component is being modified

  (if (not (normal_state s))
      s

  (if (not (normal_state sBV))
      sBV

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule e (prodn (tag 'value_modifiers 'm)
                     (tag 'component_selectors 's)))
      (GPF_modifiers sBV (subtree e 'component_selectors) c s (sub1 n) x)

  (if (rule e (prodn (tag 'component_selectors 's)
                     (list 'DOT (tag 'IDENTIFIER 'fn))))
      (GPF_record_get sBV
                      (allocate 'result~
                                (marked 'field_name
                                        (gname (subtree e 'IDENTIFIER)))
                                s)
                      s)

  (if (rule e (prodn (tag 'component_selectors 's)
                     (tag 'arg_list 'd)))
      (GPF_select_op sBV (GPF_adp (subtree e 'arg_list) c s (sub1 n) x) s)

  (if (rule e (prodn (tag 'value_modifiers 'm)
                     (tag 'range 'r)))
      (GPF_modifiers sBV (subtree e 'range) c s (sub1 n) x)

  (if (rule e (prodn (tag 'range 'r)
                     (list 'OPEN_PAREN (tag 'range_limits 'r2)
                           'CLOSE_PAREN)))
      (GPF_modifiers sBV (subtree e 'range_limits) c s (sub1 n) x)

  (if (rule e (prodn (tag 'range_limits 'r)
                     (list (tag 'expression 'lo) 'DOT_DOT
                           (tag 'expression 'hi))))
      (GPF_subsequence_get sBV
                           (GPF (subtree_i e 'expression 1) c s (sub1 n) x)
                           (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
                           s)

  (if (rule e (prodn (tag 'value_modifiers 'm)
                     (tag 'value_alterations 'a)))
      (GPF_modifiers sBV (subtree e 'value_alterations) c s (sub1 n) x)

  (if (rule e (prodn (tag 'value_alterations 'a)
                     (list 'WITH 'OPEN_PAREN
```

```
                           (tag 'component_alterations_list 'al)
                           'CLOSE_PAREN)))
      (GPF_modifiers sBV (subtree e 'component_alterations_list)
                     c s (sub1 n) x)

  (if (rule e (prodn (tag 'component_alterations_list 'al)
                     (tag 'component_alterations 'a)))
      (GPF_modifiers sBV (subtree e 'component_alterations) c s (sub1 n) x)

  (if (rule e (prodn (tag 'component_alterations_list 'al)
                     (list (tag 'component_alterations_list 'al2)
                           'SEMI_COLON (tag 'component_alterations 'a))))
      (GPF_modifiers (GPF_modifiers sBV
                                    (subtree e 'component_alterations_list)
                                    c s (sub1 n) x)
                     (subtree e 'component_alterations) c s (sub1 n) x)

  (if (rule e (prodn (tag 'component_alterations 'as)
                     (list (tag 'opt_each_clause 'e)
                           (tag 'component_assignment 'a))))
      (if (each_clausep (subtree e 'opt_each_clause))
          (let ((sVS (GPF_bound_values (subtree e 'opt_each_clause) c s x)))
            (if (normal_state sVS)
                (GPF_each (bound_id (subtree e 'opt_each_clause))
                          (result~ sVS) sBV (subtree e 'component_assignment)
                          c s (sub1 n) x)
              sVS))
        (GPF_modifiers sBV (subtree e 'component_assignment) c s (sub1 n) x))

  (if (rule e (prodn (tag 'component_alterations 'as)
                     (list (tag 'opt_each_clause 'e)
                           (tag 'component_creation 'c))))
      (if (each_clausep (subtree e 'opt_each_clause))
          (let ((sVS (GPF_bound_values (subtree e 'opt_each_clause) c s x)))
            (if (normal_state sVS)
                (GPF_each (bound_id (subtree e 'opt_each_clause))
                          (result~ sVS) sBV (subtree e 'component_creation)
                          c s (sub1 n) x)
              sVS))
        (GPF_modifiers sBV (subtree e 'component_creation) c s (sub1 n) x))

  (if (rule e (prodn (tag 'component_alterations 'as)
                     (list (tag 'opt_each_clause 'e)
                           (tag 'component_deletion 'd))))
      (if (each_clausep (subtree e 'opt_each_clause))
          (let ((sVS (GPF_bound_values (subtree e 'opt_each_clause) c s x)))
            (if (normal_state sVS)
                (GPF_each (bound_id (subtree e 'opt_each_clause))
                          (result~ sVS) sBV (subtree e 'component_deletion)
                          c s (sub1 n) x)
              sVS))
        (GPF_modifiers sBV (subtree e 'component_deletion) c s (sub1 n) x))

  (if (rule e (prodn (tag 'component_assignment 'a)
                     (list (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (GPF_put_op sBV
                  (GPF_selectors (subtree e 'selector_list)
                                 c s (sub1 n) x)
                  (GPF (subtree e 'expression) c s (sub1 n) x)
                  s)

  (if (rule e (prodn (tag 'component_creation 'c)
                     (list 'BEFORE (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x))
            (sU (GPF (subtree e 'expression) c s (sub1 n) x)))
        (GPF_put_op sBV (rcdr sS)
                    (GPF_Gseq_insert_before (GPF_select_op sBV (rcdr sS) s)
```

```
                                            (rcar sS) sU s)
                    s))

  (if (rule e (prodn (tag 'component_creation 'c)
                     (list 'BEHIND (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x))
            (sU (GPF (subtree e 'expression) c s (sub1 n) x)))
        (GPF_put_op sBV (rcdr sS)
                    (GPF_Gseq_insert_behind (GPF_select_op sBV (rcdr sS) s)
                                            (rcar sS) sU s)
                    s))

  (if (rule e (prodn (tag 'component_creation 'c)
                     (list 'INTO (tag 'selector_list 's)
                           'COLON_EQUAL (tag 'expression 'e))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x))
            (sU (GPF (subtree e 'expression) c s (sub1 n) x)))
        (GPF_put_op sBV (rcdr sS)
                    (GPF_Gmap_insert (GPF_select_op sBV (rcdr sS) s)
                                     (rcar sS) sU s)
                    s))

  (if (rule e (prodn (tag 'component_deletion 'd)
                     (list 'SEQOMIT (tag 'selector_list 's))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x)))
        (GPF_put_op sBV (rcdr sS)
                    (GPF_Gseqomit (GPF_select_op sBV (rcdr sS) s)
                                  (rcar sS) s)
                    s))

  (if (rule e (prodn (tag 'component_deletion 'd)
                     (list 'MAPOMIT (tag 'selector_list 's))))
      (let ((sS (GPF_selectors (subtree e 'selector_list) c s (sub1 n) x)))
        (GPF_put_op sBV (rcdr sS)
                    (GPF_Gmapomit (GPF_select_op sBV (rcdr sS) s)
                                  (rcar sS) s)
                    s))

    (mark_state_indeterminate s))))))))))))))))))))))

  ( (lessp (count n)) ))



; ************************************
;  Name references and function calls
; ************************************

(defn GPF_apply_fun (fn adp acp sn s n x)
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (let ((h (car (ref fn sn x)))  ; scope fn is declared in
          (u (cdr (ref fn sn x)))) ; the function declaration
      (if (equal (kind u) 'function)
          (let ((fs (formal_dargs u)))                    ; formals
          (let ((ds (if (equal (length fs) 0) nil adp))   ; actuals
                (ss (if (equal (length fs) 0) adp nil)))  ; selectors
            (GPF_select_op (GP_procedure_call fn ds acp sn s (sub1 n) x)
                           ss s)))
      (if (equal (kind u) 'constant)
          (if (equal acp nil)
              (GPF_select_op
                 (GPF_retype_result~ (GPF (constant_body u) h s (sub1 n) x)
                                     (type_desc (result_type u) h nil x))
                 adp s)
            (set_condition s 'routineerror))
        (set_condition s 'routineerror)))))
  ( (lessp (count n)) ))
```

```
(defn GPF_apply (fn adp acp sn s n x)
  ; adp is a list of states
  (if (state_componentp fn s)
      (if (equal acp nil)
          (GPF_apply_var fn s adp)
        (set_condition s 'routineerror))
  (if (equal fn 'false)
      (if (equal acp nil)
          (GPF_select_op (GPF_false s) adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'true)
      (if (equal acp nil)
          (GPF_select_op (GPF_true s) adp s)
        (set_condition s 'routineerror))
  (if (type_name_expp fn adp sn x)
      (if (equal acp nil)
          (GPF_type_name_arg fn sn s x)
        (set_condition s 'routineerror))
  (if (equal fn 'domain)
      (if (equal acp nil)
          (GPF_std_domain adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'first)
      (if (equal acp nil)
          (GPF_std_first adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'initial)
      (if (equal acp nil)
          (GPF_std_initial adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'last)
      (if (equal acp nil)
          (GPF_std_last adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'lower)
      (if (equal acp nil)
          (GPF_std_lower adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'max)
      (if (equal acp nil)
          (GPF_std_max adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'min)
      (if (equal acp nil)
          (GPF_std_min adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'nonfirst)
      (if (equal acp nil)
          (GPF_std_nonfirst adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'nonlast)
      (if (equal acp nil)
          (GPF_std_nonlast adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'null)
      (if (equal acp nil)
          (GPF_std_null adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'ord)
      (if (equal acp nil)
          (GPF_std_ord adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'pred)
      (if (equal acp nil)
          (GPF_std_pred adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'range)
      (if (equal acp nil)
          (GPF_std_range adp s)
```

```
        (set_condition s 'routineerror))
  (if (equal fn 'scale)
      (if (equal acp nil)
          (GPF_std_scale adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'size)
      (if (equal acp nil)
          (GPF_std_size adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'succ)
      (if (equal acp nil)
          (GPF_std_succ adp s)
        (set_condition s 'routineerror))
  (if (equal fn 'upper)
      (if (equal acp nil)
          (GPF_std_upper adp s)
        (set_condition s 'routineerror))
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (GPF_apply_fun fn adp acp sn s (sub1 n) x))))))))))))))))))))))
  ( (lessp (count n)) ))


(defn GPF_list (es c s n x)
  (if (nlistp es)
      nil
  (if (zerop (fix n))
      (list (mark_state_indeterminate s))
    (cons (GPF (car es) c s (sub1 n) x)
          (GPF_list (cdr es) c s (sub1 n) x))))
  ( (lessp (count n)) ))

(defn GPF (e c s n x)
  ; The meta-function GPF(e,c,s,n,x) gives the state that results when the
  ; expression e is interpreted for at most n steps, in the context of scope
  ; c, initial state s, and Gypsy sentence x.
  ;
  ; The domain and range of GPF(e,c,s,n,x) are as follows:
  ;
  ; e is the parse tree of an expression which describes a mechanism.
  ; c is the (litatom) name of the Gypsy scope in which e appears.
  ; s is  the marked, initial state on which the expression mechanism
  ;   e begins to operate.
  ; n is the maximum number of steps that the expression mechanism is
  ;   allowed to operate.
  ; x is  the  parse  tree  of the program description sentence which
  ;   defines the library which  contains  the  declarations  of  the
  ;   Gypsy units which are referred to by e.
  ; gPF(e,c,s,n,x) is the marked, final state which results from operating the
  ;                mechanism on the initial state s for at most n steps.  If
  ;                the mechanism has not completed all of its operations in n
  ;                steps, then the final state is marked as indeterminate.

  (if (not (normal_state s))
      s

  (if (zerop (fix n))
      (mark_state_indeterminate s)


  ; ************************
  ;
  ;    <expression> ::= ...
  ;
  ; ************************

  (if (rule e (prodn (tag 'expression 'e)
                     (tag 'modified_primary_value 'm)))
      (GPF (subtree e 'modified_primary_value) c s (sub1 n) x)
```

```
(if (rule e (prodn (tag 'expression 'e)
                   (list 'ALL (tag 'bound_expression 'b))))
    (let ((sVS (GPF_bound_values e c s x)))
      (if (normal_state sVS)
          (GPF_all (bound_id (subtree e 'bound_expression))
                   (result~ sVS) (cdr_quantified_exp e)
                   c s (sub1 n) x)
        sVS))

(if (rule e (prodn (tag 'expression 'e)
                   (list 'SOME (tag 'bound_expression 'b))))
    (let ((sVS (GPF_bound_values e c s x)))
      (if (normal_state sVS)
          (GPF_some (bound_id (subtree e 'bound_expression))
                    (result~ sVS) (cdr_quantified_exp e)
                    c s (sub1 n) x)
        sVS))

(if (rule e (prodn (tag 'expression 'e)
                   (list (tag 'unary_operator 'op) (tag 'expression 'e2))))
    (GPF_apply_unary_op (subtree e 'unary_operator)
                        (GPF (subtree e 'expression) c s (sub1 n) x)
                        s)

(if (rule e (prodn (tag 'expression 'e)
                   (list (tag 'expression 'e1) (tag 'binary_operator 'op)
                         (tag 'expression 'e2))))
    (GPF_apply_binary_op (subtree e 'binary_operator)
                         (GPF (subtree_i e 'expression 1) c s (sub1 n) x)
                         (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
                         s)


; ********************************
;
;    <modified primary value> ::=
;
; ********************************

(if (rule e (prodn (tag 'modified_primary_value 'm)
                   (tag 'primary_value 'p)))
    (GPF (subtree e 'primary_value) c s (sub1 n) x)

(if (rule e (prodn (tag 'modified_primary_value 'm)
                   (list (tag 'modified_primary_value 'm2)
                         (tag 'value_modifiers 'vm))))
    (if (fn_call_formp e)
        (GPF_apply (object_name (subtree e 'modified_primary_value))
                   (GPF_adp (arg_list (subtree e 'value_modifiers))
                            c s (sub1 n) x)
                   nil c s (sub1 n) x)
      (GPF_modifiers (GPF (subtree e 'modified_primary_value)
                          c s (sub1 n) x)
                     (subtree e 'value_modifiers) c s (sub1 n) x))

(if (rule e (prodn (tag 'modified_primary_value 'm)
                   (list (tag 'modified_primary_value 'm2)
                         (tag 'actual_condition_parameters 'cp))))
    (if (fn_call_formp e)
        (GPF_apply (object_name (subtree e 'modified_primary_value))
                   (GPF_adp (arg_list (subtree e 'modified_primary_value))
                            c s (sub1 n) x)
                   (actual_cargs e)
                   c s (sub1 n) x)
      (set_condition s 'routineerror))


; **********************
;
```

```
;    <primary value> ::=
;
; **********************

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'literal_value 'l)))
    (GPF (subtree e 'literal_value) c s (sub1 n) x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'set_or_sequence_value 's)))
    (GPF (subtree e 'set_or_sequence_value) c s (sub1 n) x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'ENTRY_VALUE 'e)))
    (GPF (subtree e 'ENTRY_VALUE) c s (sub1 n) x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'IDENTIFIER 'on)))
    (GPF (subtree e 'IDENTIFIER) c s (sub1 n) x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (tag 'if_expression 'i)))
    (GPF (subtree e 'if_expression) c s (sub1 n) x)

(if (rule e (prodn (tag 'primary_value 'p)
                   (list 'OPEN_PAREN (tag 'expression 'e) 'CLOSE_PAREN)))
    (GPF (subtree e 'expression) c s (sub1 n) x)


; -----------------------------------------------------------------------
;
;  From here down to parse tree leaves, clauses are in alphabetical order
;  by the left-hand side of the productions.  Everything that is a parse
;  tree for an expression should be covered.
;
; -----------------------------------------------------------------------


; **********************
;
;    <constant body> ::=
;
; **********************

(if (rule e (prodn (tag 'constant_body 'b)
                   (tag 'expression 'e)))
    (GPF (subtree e 'expression) c s (sub1 n) x)

(if (rule e (prodn (tag 'constant_body 'b)
                   'PENDING))
    (mark_state_indeterminate s)


; **********************
;
;    <if expression> ::=
;
; **********************

(if (rule e (prodn (tag 'if_expression 'i)
                   (list 'IF (tag 'expression 'b) 'THEN
                         (tag 'expression 'p)
                         (tag 'if_expression_else_part 'e))))

    ; Note:  this does not require all potential value expressions to be the
    ; same type or all boolean expressions to be boolean.
    (let ((bv (GPF_type_check (boolean_desc)
                              (GPF (subtree_i e 'expression 1)
                                   c s (sub1 n) x))))
```

```
      (if (not (normal_state bv))
          bv
        (if (Gtruep (result~ bv))
            (GPF (subtree_i e 'expression 2) c s (sub1 n) x)
          (GPF (if_else_exp (subtree e 'if_expression_else_part))
               c s (sub1 n) x))))


; **********************
;
;    <literal value> ::=
;
; **********************

(if (rule e (prodn (tag 'literal_value 'l)
                   (tag 'CHARACTER_VALUE 'ch)))
    (GPF (subtree e 'CHARACTER_VALUE) c s (sub1 n) x)

(if (rule e (prodn (tag 'literal_value 'l)
                   (tag 'number 'n)))
    (GPF (subtree e 'number) c s (sub1 n) x)

(if (rule e (prodn (tag 'literal_value 'l)
                   (tag 'STRING_VALUE 's)))
    (GPF (subtree e 'STRING_VALUE) c s (sub1 n) x)


; ****************
;
;    <number> ::=
;
; ****************

(if (rule e (prodn (tag 'number 'n)
                   (tag 'DIGIT_LIST 's)))
    (GPF_minteger e s)

(if (rule e (prodn (tag 'number 'n)
                   (list (tag 'base 'b) (tag 'DIGIT_LIST 's))))
    (GPF_minteger e s)


; ****************************************
;
;    <pre-computable label expression> ::=
;
; ****************************************

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                   (tag 'number 'n)))
    (GPF (subtree e 'number) c s (sub1 n) x)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                   (list 'MINUS (tag 'number 'n))))
    (GPF_apply_unary_op (mk_unary_operator 'MINUS)
                        (GPF (subtree e 'number) c s (sub1 n) x)
                        s)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                   (tag 'CHARACTER_VALUE 'ch)))
    (GPF (subtree e 'CHARACTER_VALUE) c s (sub1 n) x)

(if (rule e (prodn (tag 'pre_computable_label_expression 'p)
                   (tag 'IDENTIFIER 'i)))
    (GPF (subtree e 'IDENTIFIER) c s (sub1 n) x)


; *******************************
;
```

```
  ;    <set or sequence value> ::=
  ;
  ; ********************************

  (if (rule e (prodn (tag 'set_or_sequence_value 's)
                     (list 'OPEN_PAREN (tag 'set_or_seq_mark 'm)
                           (tag 'element_list 'e) 'CLOSE_PAREN)))
      (GPF_Gset_or_seq (subtree e 'set_or_seq_mark)
                       (GPF_element_list (subtree e 'element_list)
                                         c s (sub1 n) x)
                       (GPF_element_type (subtree e 'element_list)
                                         c s (sub1 n) x)
                       s)

  (if (rule e (prodn (tag 'set_or_sequence_value 's)
                     (tag 'range 'r)))
      (GPF_Gset_or_seq nil
                       (GPF_element_list (subtree e 'range) c s (sub1 n) x)
                       (GPF_element_type (subtree e 'range) c s (sub1 n) x)
                       s)


  ; *********************
  ;
  ;    PARSE TREE LEAVES
  ;
  ; *********************

  (if (character_valuep e)
      (GPF_Gchar e s)

  (if (digit_listp e)
      (GPF_minteger e s)

  (if (entry_valuep e)
      (GPF_apply_var (entry_name e) s nil)

  (if (identifierp e)
      (GPF_apply (gname e) nil nil c s (sub1 n) x)

  (if (string_valuep e)
      (GPF_Gstring_seq e s)

    (mark_state_indeterminate s))))))))))))))))))))))))))))))))))

  ( (lessp (count n)) ))


; ***************************************************************************
;
;                       THE PROCEDURAL INTERPRETER
;
; ***************************************************************************

; ==========================================================
;  Computation of Actual Data Parameters for Procedure Call
; ==========================================================

(defn GP_parg (e c s n x)
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (let ((e2 (mk_name_expression e)))
      (if (rule e2 (prodn (tag 'name_expression 'e)
                          (tag 'IDENTIFIER 'i)))
          (let ((vn (gname (subtree e2 'IDENTIFIER))))
            (let ((r (GPF_apply_var vn s nil)))
              (if (normal_state r)
                  (allocate 'result~ (name_exp vn nil) s)
```

```
                r)))

      (if (rule e2 (prodn (tag 'name_expression 'e)
                          (list (tag 'IDENTIFIER 'i)
                                (tag 'selector_list 'ss))))
          (let ((vn (gname (subtree e2 'IDENTIFIER)))
                (ss (GPF_selectors (subtree e2 'selector_list)
                                   c s (sub1 n) x)))
            (let ((r (GPF_apply_var vn s ss)))
              (if (normal_state r)
                  (allocate 'result~ (name_exp vn (result~_list ss)) s)
                r)))

        (GPF e c s (sub1 n) x)))))

  ( (lessp (count n)) ))

(defn GP_parg_list (as c s n x)
  (if (nlistp as)
      nil
  (if (zerop (fix n))
      (list (mark_state_indeterminate s))
    (cons (GP_parg (car as) c s (sub1 n) x)
          (GP_parg_list (cdr as) c s (sub1 n) x))))
  ( (lessp (count n)) ))



; ====================
;  The Procedure Call
; ====================

(defn GP_procedure_call (pn Sadp acp c s n x)
  ; pn is the called procedure name
  ; Sadp is the list of states resulting from evaluation of actual data
  ;      parameters
  ; acp is the list of actual condition paramters
  ; entry (normal_state s)
  (if (zerop (fix n))
      (mark_state_indeterminate s)
    (let ((h (car (ref pn c x)))
          (u (cdr (ref pn c x))))
      (if (or (equal (kind u) 'procedure)
              (equal (kind u) 'function))
          (let ((s1 (GP_call_state u h Sadp acp s x)))
            (if (normal_state s1)
                (GP_map_call_effects
                  (GP_procedure_body (procedure_body u) h s1 (sub1 n) x)
                  u (result~_list Sadp) acp c s n x)
              (map_cond_effects (cond~ s1) (formal_cargs u) acp
                                (marked (mark s1) (map s)))))
        (set_condition s 'routineerror)))) ; indeterminate?
  ( (lessp (count n)) ))

(defn GP_procedure_body (m c s n x)

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule m (prodn (tag 'procedure_body 'b)
                     'PENDING))
      (mark_state_indeterminate s)

  (if (rule m (prodn (tag 'procedure_body 'b)
                     (list 'BEGIN
                           (tag 'external_operational_specification 'es)
                           (tag 'opt_internal_environment 'iv)
                           (tag 'opt_keep_specification 'k)
                           (tag 'opt_internal_statements 'st)
```

```
                           'END)))
      (let ((r (GP_deallocate_locals
                (GP (subtree m 'opt_internal_statements)
                    c
                    (GP_set_keep
                     (keep_spec m)
                     (GP_locals (subtree m 'opt_internal_environment)
                                c
                                (GP_set_entry (prec m) c s n x)
                                (sub1 n) x)
                     c n x)
                    (sub1 n) x))))
        (if (indeterminate r)
            r
        (if (or (equal (cond~ r) 'normal)
                (conditionp (cond~ r) r))
            (GP_set_exit (exit_spec m) c r n x)
        (if (equal (cond~ r) 'leave)
            ; leave statement was not in a loop
            (set_condition r 'routineerror)
          ; condition signalled was not a forward condition; we are not
          ; allowed to signal routineerror
          (mark_state_indeterminate r)))))

  (mark_state_indeterminate s))))

  ( (lessp (count n)) ))

(defn GP_locals (m c s n x)

  (if (not (normal_state s))
      s

  (if (zerop n)
      (mark_state_indeterminate s)

  (if (rule m (prodn (tag 'opt_internal_environment 'iv)
                     'empty))
      s

  (if (rule m (prodn (tag 'opt_internal_environment 'iv)
                     (tag 'internal_environment 'iv2)))
      (GP_locals (subtree m 'internal_environment) c s (sub1 n) x)

  (if (rule m (prodn (tag 'internal_environment 'iv)
                     (tag 'internal_data_or_condition_objects 'iv2)))
      (GP_locals (subtree m 'internal_data_or_condition_objects)
                 c s (sub1 n) x)

  (if (rule m (prodn (tag 'internal_environment 'iv)
                     (list (tag 'internal_environment 'iv2)
                           (tag 'internal_data_or_condition_objects 'iv3))))
      (GP_locals (subtree m 'internal_data_or_condition_objects) c
                 (GP_locals (subtree m 'internal_environment) c s (sub1 n) x)
                 (sub1 n) x)

  (if (rule m (prodn (tag 'internal_data_or_condition_objects 'iv)
                     (list (tag 'access_specification 'a)
                           (tag 'identifier_list 'is) 'COLON
                           (tag 'type_specification 'ts)
                           (tag 'opt_internal_initial_value 'v)
                           'SEMI_COLON)))
      (let ((ie (internal_initial_value_exp m)))
      (let ((iv (if (equal ie nil)
                    ie
                  (GPF ie c s (sub1 n) x))))
        (GP_bind_locals (access m)
                        (id_list (subtree m 'identifier_list))
                        (type_desc (subtree m 'type_specification) c nil x)
```

```
                        iv s)))

  (if (rule m (prodn (tag 'internal_data_or_condition_objects 'iv)
                     (list 'COND (tag 'identifier_list 'is) 'SEMI_COLON)))

      (GP_local_conds (id_list (subtree m 'identifier_list)) s)

    (mark_state_indeterminate s)))))))))

  ( (lessp (count n)) ))



; ****************
;  Case Statement
; ****************

(defn GP_case_body (k m c s n x)

  (if (not (normal_state s))
      s

  (if (zerop (fix n))
      (mark_state_indeterminate s)

  (if (rule m (prodn (tag 'case_composition_body 'b)
                     'empty))
      s

  (if (rule m (prodn (tag 'case_composition_body 'b)
                     (list 'ELSE 'COLON (tag 'opt_internal_statements 'ss))))
      (GP (subtree m 'opt_internal_statements) c s (sub1 n) x)

  (if (rule m (prodn (tag 'case_composition_body 'b)
                     (list 'IS (tag 'case_labels 'cs) 'COLON
                           (tag 'opt_internal_statements 'ss)
                           (tag 'case_composition_body 'b2))))
      (let ((s1 (GPF_Gin k
                         (GPF_Gset (GPF_list (case_labels m) c s (sub1 n) x)
                                   (base_type (type (result~ k)))
                                   s)
                         s)))
        (if (normal_state s1)
            (if (Gtruep (result~ s1))
                (GP (subtree m 'opt_internal_statements) c s (sub1 n) x)
              (GP_case_body k (subtree m 'case_composition_body)
                            c s (sub1 n) x))
          s1))

    (mark_state_indeterminate s))))))

  ( (lessp (count n)) ))



; *****************************************************************
;
;                      HANDLING CONDITIONS
;
;  <opt_condition_handlers> ::= <empty> | WHEN <opt handler list>
;  <opt_handler_list> ::= empty | <handler_list>
;  <handler_list> ::= <handler> { <handler> }
;
; *****************************************************************

(defn GP_cond (m c s n x)
  (if (or (indeterminate s) (condition_normal s))
      s
  (if (or (zerop n)
```

```
          (not (condition_labels_ok (handler_labels m) s)))
      (mark_state_indeterminate s)
    (let ((h (handler m (cond~ s))))
      (if (equal h nil)
          s
        (GP h c (set_condition s 'normal) (sub1 n) x)))))
  ( (lessp (count n)) ))



;; **** The strange commented characters in the following function allow GNU
;;      Emacs to count parentheses correctly.


(defn GP (m c s n x)
  ; The meta-function GP (m,c,s,n,x) gives the state that results
  ; when the program fragment m is interpreted for at most n steps, in the
  ; context of scope c, initial state s, and Gypsy sentence x.
  ;
  ; The domain and range of GP (m,c,s,n,x) are as follows:
  ;
  ; m is the parse tree which describes a computer program mechanism.
  ; c is the (litatom) name of the Gypsy scope in which m appears.
  ; s is the marked, initial state on which the mechanism m begins to operate.
  ; n is the maximum number of steps  the  mechanism  is  allowed  to perform.
  ; x is  the  parse  tree  of the program description sentence which
  ;   defines the library which  contains  the  declarations  of  the
  ;   Gypsy units which are referred to by m.
  ; GP(m,c,s,n,x) is  the  marked,  final  state which results from operating
  ;               the mechanism on the initial state s for at most n steps.
  ;               If the mechanism  has  not  completed all of its operation
  ;               in n steps, then the final state is marked as indeterminate.

  (if (not (normal_state s))
      s

  (if (zerop (fix n))
      (mark_state_indeterminate s)


  ; ***************************************************
  ;
  ;                  STATEMENT LISTS
  ;
  ;  <statement list> ::= <statement> {; <statement> }
  ;
  ; ***************************************************

  (if (rule m (prodn (tag 'statement_list 'ss)
                     (tag 'statement 's)))
      (GP (subtree m 'statement) c s (sub1 n) x)

  (if (rule m (prodn (tag 'statement_list 'ss)
                     (list (tag 'statement_list 'ss2) 'SEMI_COLON
                           (tag 'statement 's))))
      (GP (subtree m 'statement)
          c
          (GP (subtree m 'statement_list) c s (sub1 n) x)
          (sub1 n)
          x)

  ; **********************************************************************
  ;
  ;  <statement> ::=   <procedural statement> | <procedure composition rule>
  ;                  | <assert specification>
  ;
  ; **********************************************************************

  (if (rule m (prodn (tag 'statement 's)
```

```
                   (tag 'procedural_statement 's2)))
    (GP (subtree m 'procedural_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'statement 's)
                   (tag 'procedure_composition_rule 's2)))
    (GP (subtree m 'procedure_composition_rule) c s (sub1 n) x)

(if (rule m (prodn (tag 'statement 's)
                   (tag 'assert_specification 's2)))
    (GP (subtree m 'assert_specification) c s (sub1 n) x)


; ************************************************************
;
;                  PROCEDURAL STATEMENT
;
;  <procedural statement> ::=   <assignment statement>
;                             | <leave statement>
;                             | <move statement>
;                             | <new statement>
;                             | <procedure statement>
;                             | <remove statement>
;                             | <signal statement>
;
; ************************************************************

(if (rule m (prodn (tag 'procedural_statement 's)
                   (tag 'assignment_statement 's2)))
    (GP (subtree m 'assignment_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
                   (tag 'leave_statement 's2)))
    (GP (subtree m 'leave_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
                   (tag 'move_statement 's2)))
    (GP (subtree m 'move_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
                   (tag 'new_statement 's2)))
    (GP (subtree m 'new_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
                   (tag 'procedure_statement 's2)))
    (GP (subtree m 'procedure_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
                   (tag 'remove_statement 's2)))
    (GP (subtree m 'remove_statement) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedural_statement 's)
                   (tag 'signal_statement 's2)))
    (GP (subtree m 'signal_statement) c s (sub1 n) x)


; *********************
;
;  ASSIGNMENT STATEMENT
;
; *********************

(if (rule m (prodn (tag 'assignment_statement 's)
                   (list (tag 'name_expression 'n) 'COLON_EQUAL
                         (tag 'expression 'e))))
    (GP_assign (GP_parg (subtree m 'name_expression) c s (sub1 n) x)
               (GPF (subtree m 'expression) c s (sub1 n) x)
               s c n x)
```

```
; ****************
;
;  LEAVE STATEMENT
;
; ****************

(if (rule m (prodn (tag 'leave_statement 's)
                   'LEAVE))
    (set_condition s 'leave)


; ************************************************************************
;
;                             MOVE STATEMENT
;
;  <move_statement> ::= MOVE <removable component> <component destination>
;  <component destination> ::=   <new dynamic variable component>
;                              | TO <sequence element name expression>
;
; ************************************************************************
                                                                     ; |<<
(if (rule m (prodn (tag 'move_statement 's)
                   (list 'MOVE (tag 'removable_component 'c)
                         (tag 'component_destination 'd))))
    (let ((e (remove_exp_arg m)))
      (GP_move (if (equal e nil) nil (GPF e c s (sub1 n) x))
               (GP_parg (remove_name_arg m) c s (sub1 n) x)
               (subtree m 'component_destination)
               (GP_parg (new_name_arg m) c s (sub1 n) x)
               c s n x))


; **********************************************************************
;
;                             NEW STATEMENT
;
;  <new_statement> ::= NEW <expression> <new dynamic variable component>
;
; **********************************************************************

(if (rule m (prodn (tag 'new_statement 's)
                   (list 'NEW (tag 'expression 'e)
                         (tag 'new_dynamic_variable_component 'dc))))
    (GP_new (subtree m 'new_dynamic_variable_component)
            (GPF (subtree m 'expression) c s (sub1 n) x)
            (GP_parg (new_name_arg m) c s (sub1 n) x)
            c s n x)


; ***************
;
;  PROCECURE CALL
;
; ***************

(if (rule m (prodn (tag 'procedure_statement 's)
                   (list (tag 'IDENTIFIER 'pn)
                         (tag 'arg_list 'dp)
                         (tag 'opt_actual_condition_parameters 'cp))))
    (GP_procedure_call (gname (subtree m 'identifier))
                       (GP_parg_list (actual_dargs m) c s (sub1 n) x)
                       (actual_cargs m)
                       c s (sub1 n) x)


; *********************************************************
;
;                      REMOVE STATEMENT
;
```

```
;  <remove_statement> ::= REMOVE <removable component>
;  <removable component> ::=
;           ELEMENT <expression> FROM SET <name expression>
;         | <name expression>
;
; ********************************************************
                                                              ; |
(if (rule m (prodn (tag 'remove_statement 's)
                   (list 'REMOVE (tag 'removable_component 'c))))
    (let ((e (remove_exp_arg m)))
      (GP_remove (if (equal e nil) nil (GPF e c s (sub1 n) x))
                 (GP_parg (remove_name_arg m) c s (sub1 n) x)
                 c s n x))


; ******************
;
;  SIGNAL STATEMENT
;
; ******************

(if (rule m (prodn (tag 'signal_statement 's)
                   (list 'SIGNAL (tag 'IDENTIFIER 'c))))
    (if (conditionp (gname (subtree m 'IDENTIFIER)) s)
        (set_condition s (gname (subtree m 'IDENTIFIER)))
      (mark_state_indeterminate s))


; **********************************************************************
;
;                       PROCEDURE COMPOSITION
;
;  <procedure composition> ::   <if_composition> | <case_composition>
;                             | <loop_composition> | <begin_composition>
;
; **********************************************************************
                                                                   ; |
(if (rule m (prodn (tag 'procedure_composition_rule 's)
                   (tag 'if_composition 's2)))
    (GP (subtree m 'if_composition) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedure_composition_rule 's)
                   (tag 'case_composition 's2)))
    (GP (subtree m 'case_composition) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedure_composition_rule 's)
                   (tag 'loop_composition 's2)))
    (GP (subtree m 'loop_composition) c s (sub1 n) x)

(if (rule m (prodn (tag 'procedure_composition_rule 's)
                   (tag 'begin_composition 's2)))
    (GP (subtree m 'begin_composition) c s (sub1 n) x)


; ******************************************************************
;
;                         IF COMPOSITION
;
;  <if composition> ::=
;         IF <boolean expression> THEN <opt internal statements>
;           <if composition else part>
;           <opt condition handlers>
;         END
;
;  <if composition else part> ::=
;           <empty>
;         | ELSE <opt internal statements>
;         | ELIF <boolean expression> THEN <opt internal statements>
;                <if composition else part>
```

```
;
; *******************************************************************

(if (rule m (prodn (tag 'if_composition 's)
                   (list 'IF (tag 'expression 'b) 'THEN
                         (tag 'opt_internal_statements 'ss)
                         (tag 'if_composition_else_part 'ep)
                         (tag 'opt_condition_handlers 'cs) 'END)))
    (let ((bv (GPF_type_check (boolean_desc)
                              (GPF (subtree m 'expression)
                                   c s (sub1 n) x)))
          (ep (if_statement_else_part m)))
      (GP_cond (subtree m 'opt_condition_handlers) c
               (if (normal_state bv)
                   (if (Gtruep (result~ bv))
                       (GP (subtree m 'opt_internal_statements)
                           c s (sub1 n) x)
                     (if (equal ep nil)
                         s
                       (GP ep c s (sub1 n) x)))
                 bv)
               (sub1 n) x))


; **************************************************************************
;
;                            CASE COMPOSITION
;
;  <case composition> ::= CASE <label expression>
;                          { IS <case labels> : [ <internal statements> ] }
;                          [ ELSE             : [ <internal statements> ] ]
;                          [ <condition handlers> ]
;                         END
;
; **************************************************************************

(if (rule m (prodn (tag 'case_composition 's)
                   (list 'CASE (tag 'expression 'e)
                         (tag 'case_composition_body 'b)
                         (tag 'opt_condition_handlers 'c) 'END)))
    (let ((k (GPF (subtree m 'expression) c s (sub1 n) x)))
    (let ((r (GP_case_label_check k
                                  (GPF_list (case_labels m) c s (sub1 n) x)
                                  s)))
      (GP_cond (subtree m 'opt_condition_handlers) c
               (if (normal_state r)
                   (GP_case_body k (subtree m 'case_composition_body)
                                 c s (sub1 n) x)
                 r)
               (sub1 n) x)))


; *******************************************************
;
;                    LOOP COMPOSITION
;
;  <loop composition> ::= LOOP [ <internal statements> ]
;                              [ <condition handlers> ]
;                         END
;
; *******************************************************

(if (rule m (prodn (tag 'loop_composition 's)
                   (list 'LOOP (tag 'opt_internal_statements 'ss)
                         (tag 'opt_condition_handlers 'c) 'END)))
    (let ((p1 (GP (subtree m 'opt_internal_statements) c s (sub1 n) x)))
      (GP_cond (subtree m 'opt_condition_handlers) c
               (if (condition_non_normal p1)
                   (reset_leave_to_normal p1)
```

```
                 (GP m c p1 (sub1 n) x))
               (sub1 n) x))


; ********************************************************
;
;                     BEGIN COMPOSITION
;
;  <begin composition> ::= BEGIN [ <internal statements> ]
;                                [ <condition handlers> ]
;                          END
;
; ********************************************************

(if (rule m (prodn (tag 'begin_composition 's)
                   (list 'BEGIN (tag 'opt_internal_statements 'ss)
                         (tag 'opt_condition_handlers 'c) 'END)))
    (GP_cond (subtree m 'opt_condition_handlers) c
             (GP (subtree m 'opt_internal_statements) c s (sub1 n) x)
             (sub1 n) x)


; *************************************************************
;
;                      INTERNAL STATEMENTS
;
;  <opt_internal_statements> ::= <empty> | <statement list> [;]
;                                        | PENDING [;]
;
; *************************************************************

(if (rule m (prodn (tag 'opt_internal_statements 'ss)
                   'empty))
    s

(if (rule m (prodn (tag 'opt_internal_statements 'ss)
                   (list (tag 'statement_list 'ss2) 'opt_semi_colon)))
    (GP (subtree m 'statement_list) c s (sub1 n) x)

(if (rule m (prodn (tag 'opt_internal_statements 'ss)
                   (list 'PENDING 'opt_semi_colon)))
    (mark_state_indeterminate s)


; **********************
;
;  ASSERT SPECIFICATION
;
; **********************

; Whenever an assertion is encountered in executable code, it is evaluated
; and the result AND'd to the assertion-accumulator component of the map.

(if (rule m (prodn (tag 'assert_specification 's)
                   (list 'ASSERT (tag 'specification_expression 'e))))
    (GP (subtree m 'specification_expression) c s (sub1 n) x)

(if (rule m (prodn (tag 'specification_expression 'e)
                   (tag 'non_validated_specification_expression 'e2)))
    (GP_update_assert (expression_from_spec m) c s n x)

(if (or (rule m (prodn (tag 'specification_expression 'e)
                       (tag 'validated_specification_expression 'e2)))
        (rule m (prodn (tag 'specification_expression 'e)
                       (list 'OPEN_PAREN
                             (tag 'validated_specification_expression 'e2)
                             'CLOSE_PAREN))))
    (GP (subtree m 'validated_specification_expression)
        c s (sub1 n) x)
```

```
  (if (rule m (prodn (tag 'validated_specification_expression 'e)
                     (list (tag 'non_validated_specification_expression 'e2)
                           'OTHERWISE (tag 'IDENTIFIER 'i))))
      (let ((r (GPF_type_check (boolean_desc)
                               (GPF (expression_from_spec m)
                                    c s (sub1 n) x))))
        (if (normal_state r)
            (if (Gtruep (result~ r))
                (GP_record_assert (result~ r) s)
              (GP (mk_signal_stmt (subtree m 'IDENTIFIER))
                  c (GP_record_assert (result~ r) s) (sub1 n) x))
          r))

    (mark_state_indeterminate s))))))))))))))))))))))))))))))))))))

  ( (lessp (count n)) ))

))



; *********************
;  The Meta Function P
; *********************

(defn meta_P (m c s n x)
  (let ((pTm (pT m 'statement))
        (pTx (pT x 'program_description)))
    (if (equal pTm nil)
        (marked 'Statement_Syntax_Error nil)
    (if (equal pTx nil)
        (marked 'Program_Description_Syntax_Error nil)
      (GP pTm c s n pTx)))))
```

Appendix E
Type Descriptors

E.1 Arrays

SPECIFICATION In scope **sn** in sentence **x**,

```
TYPE IDENTIFIER EQUAL
     ARRAY OPEN_PAREN <type specification, sts> CLOSE_PAREN
                  OF <type_specification, cts>
```

DESCRIPTOR **array_desc(sd,cd) =**

```
                           'array
                              |
                ------------------------------
                |             |              |
           'selector_td 'component_td      'udv
                |             |              |
                sd            cd   mk_array_default(sd,cd)
```

where

sd = type_desc(sts,sn,nil,x)

cd = type_desc(cts,sn,nil,x)

provided **sd** and **cd** do not contain errors.

RECOGNIZER **root(d)='array**

E.2 Boolean

SPECIFICATION **BOOLEAN**

DESCRIPTOR **boolean_desc =**

```
            'scalar
               |
    ------------------------------
    |      |    |     |     |    |
  'tid   'sid 'crd 'tmin 'tmax 'udv
    |      |    |     |     |    |
'boolean  nil   2     0     1    0
```

RECOGNIZER **boolean_typep(d)**

E.3 Character

SPECIFICATION **CHARACTER**

DESCRIPTOR **character_desc =**

```
                  'scalar
                     |
     ------------------------------
     |       |    |     |     |    |
   'tid    'sid 'crd 'tmin 'tmax 'udv
     |       |    |     |     |    |
'character  nil  128    0    127   0
```

RECOGNIZER **character_typep(d)**

E.4 Integer

SPECIFICATION **INTEGER**

DESCRIPTOR **integer_desc =**

```
                 'integer
                     |
                ------------
                |    |     |
             'tmin 'tmax 'udv
                |    |     |
               nil  nil    0
```

RECOGNIZER **integer_typep(d)**

E.5 Mappings

SPECIFICATION In scope **sn** in sentence **x**,

```
TYPE IDENTIFIER EQUAL
     MAPPING <opt_size_limit_restriction, sls>
        FROM <type_specification, sts>
          TO <type_specification, cts>
```

DESCRIPTOR **mapping_desc(sl,sd,cd) =**

```
                              'mapping
                                  |
             --------------------------------------------
             |              |               |           |
        'max_size    'selector_td    'component_td   'udv
             |              |               |           |
            sl             sd              cd        null_map
                                                     = nil
```

where

sl = size_limit(sls)

sd = type_desc(sts,sn,nil,x)

cd = type_desc(cts,sn,nil,x)

provided **sd** and **cd** do not contain errors.

RECOGNIZER **root(d)='mapping**

E.6 Pending

SPECIFICATION In scope **sn** in sentence **x**,

```
TYPE <IDENTIFIER, tn> EQUAL PENDING
```

DESCRIPTOR **pending_desc(tid,sn) =**

```
                      'pending
                          |
                -----------------------
                |         |           |
              'tid      'sid        'udv
                |         |           |
               tid        sn       <error>
```

where

tid = gname(tn)

RECOGNIZER **root(d)='pending**

E.7 Rational

SPECIFICATION **RATIONAL**

DESCRIPTOR **rational_desc =**

```
                   'rational
                       |
                 -------------
                 |     |     |
               'tmin 'tmax 'udv
                 |     |     |
                nil   nil   rational(0,1)
```

RECOGNIZER **rational_typep(d)**

E.8 Records

SPECIFICATION In scope **sn** in sentence **x**,

TYPE IDENTIFIER EQUAL
RECORD OPEN_PAREN <fields, fts> CLOSE_PAREN

DESCRIPTOR **record_desc(fds) =**

```
                             'record
                                |
                   -----------------------
                   |                     |
              'field_tds               'udv
                   |                     |
                  fds          mk_record_default(fds)
```

where

fds = field_descs(fts,sn,nil,x)

provided **fds** does not contain errors. The parameter **fds** is a name-value mapping from field names to type descriptors.

RECOGNIZER **root(d)='record**

E.9 Scalars

SPECIFICATION In scope **sn** in sentence **x**,

```
TYPE <IDENTIFIER, tn> EQUAL <scalar_type, st>
```

DESCRIPTOR

```
scalar_desc(tid,sn,z) =
            'scalar
               |
  ----------------------------
  |    |    |     |     |    |
'tid 'sid 'crd 'tmin 'tmax 'udv
  |    |    |     |     |    |
 tid   sn   z     0    z-1   0
```

where

```
tid = gname(tn)

  z = length(scalar_value_list(st))
```

RECOGNIZER **scalar_typep(d)**

E.10 Sequences

SPECIFICATION In scope **sn** in sentence **x**,

```
TYPE IDENTIFIER EQUAL
     SEQUENCE <opt_size_limit_restriction, sls>
           OF <type_specification, cts>
```

DESCRIPTOR

```
sequence_desc(sl,cd) =
                         'sequence
                             |
              ---------------------------
              |              |             |
          'max_size   'component_td    'udv
              |              |             |
              sl             cd        null_seq = nil
```

where

```
sl = size_limit(sls)

cd = type_desc(cts,sn,nil,x)
```

provided **cd** does not contain errors.

RECOGNIZER **root(d)='sequence**

E.11 Sets

SPECIFICATION In scope **sn** in sentence **x**,

```
TYPE IDENTIFIER EQUAL
     SET <opt_size_limit_restriction, sls>
      OF <type_specification, cts>
```

DESCRIPTOR **set_desc(sl,cd) =**

```
                           'set
                             |
                ---------------------------
                |            |            |
            'max_size  'component_td    'udv
                |            |            |
                sl           cd       null_set = nil
```

where

sl = size_limit(sls)

cd = type_desc(cts,sn,nil,x)

provided **cd** does not contain errors.

RECOGNIZER **root(d)='set**

E.12 Subranges

SPECIFICATION **<IDENTIFIER, tn> <range, r>**

DESCRIPTOR **subrange_desc(d,a,b)** where

d = type_desc(tn,sn,nil,x)

a = range_min(r,sn,x)

b = range_max(r,sn,x)

The descriptor for a subrange is constructed such that

```
 tid(subrange_desc(d,a,b)) = tid(d)

 sid(subrange_desc(d,a,b)) = sid(d)

 crd(subrange_desc(d,a,b)) = crd(d)

tmin(subrange_desc(d,a,b)) = a

tmax(subrange_desc(d,a,b)) = b

 udv(subrange_desc(d,a,b)) = if udv(d) in [a..b]
                             then udv(d) else a
```

RECOGNIZER None.

References

[Aho 86] Aho, A.V., Sethi R. & Ullman, J.D.
Compilers: Principles Techniques and Tools.
Addison Wesley, 1986.

[Bevier 88] Bill Bevier.
A Library for Hardware Verification.
Internal Note 57, Computational Logic, Inc., June, 1988.
Draft.

[Boyer 89] R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore.
Functional Instantiation in First Order Logic, Report 44.
Technical Report, Computational Logic, 1717 W. 6th St., Austin, Texas, 78703, U.S.A., 1989.
To appear in the proceedings of the 1989 Workshop on Programming Logic, Programming Methodology Group, University of Goteborg.

[Boyer & Moore 79]
R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer & Moore 88]
R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[Good 78] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September, 1978.

[Good 88] Michael K. Smith, Donald I. Good, Benedetto L. DiVito.
Using the Gypsy Methodology
Computational Logic Inc., 1988.
Revised January 1988.

[Good 89a] Donald I. Good, Robert L. Akers, Lawrence M. Smith.
Report on Gypsy 2.05
Computational Logic Inc., 1989.
Revised January 10, 1989.

[Good 89b] Donald I. Good.
Mathematical Forecasting.
Technical Report CLI-47, Computational Logic, Inc., September, 1989.

[Wilding 90] Matthew Wilding.
A Mechanically-Checked Correctness Proof of a Floating-Point Search Program.
Technical Report 56, Computational Logic, Inc., May, 1990.
Draft.

[Young 88] William D. Young.
A Verified Code Generator for a Subset of Gypsy.
Technical Report 33, Computational Logic, Inc., 1988.
Ph.D. Thesis, University of Texas at Austin.

Index

Table of Contents

List of Figures

List of Tables