The Gypsy 2.0 and Gypsy 2.1 differences have been removed and placed in a separate document.)

Report on Gypsy 2.05
February 1, 1986

Donald I. Good
Robert L. Akers
Lawrence M. Smith

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

Abstract

Gypsy is a collection of methods, languages, and tools for building formally verified computing systems. Gypsy provides capabilities specifying a system, implementing it, and for using formal, logical deduction to prove important properties about the specification and the implementation of the system. The Gypsy program description language is a single, unified language that is used to express both the specification and the implementation of a computing system. This report defines the Gypsy 2.05 program description language. Gypsy 2.05 includes almost all of Gypsy 2.0 with some extensions and minor modifications.

Preface

The development of Gypsy began late in 1974, and the first report on Gypsy 1.0 was issued in August 1976. Initial attempts to use Gypsy 1.0, to define its specification and proof methods and to implement it led to a number of significant language revisions. The report on Gypsy 2.0 was issued in September 1978. Although Gypsy 2.0 extended Gypsy 1.0 in some significant ways, Gypsy 2.0 primarily was a simplification of Gypsy 1.0. In order to provide a stable implementation target, the definition of Gypsy 2.0 has remain fixed until this time. Now, based on the experience of the last several years of using and implementing Gypsy 2.0, this report describes Gypsy 2.05. Again, Gypsy 2.05 primarily is a slightly extended subset of Gypsy 2.0.

The style and organization of this report on Gypsy 2.05 is a major change from the Gypsy 2.0 report. The reason for this change is to make the report much more concise and readable. The style of presentation is informal, but precise, and the organization is from the simpler to the more complex parts of the language. Chapter 1 gives a summary of the basic Gypsy concepts. Chapters 2-7 are sufficient to specify and implement simple sequential programs. Chapters 8-11 describe the more advanced parts of Gypsy, exception conditions, dynamic objects, concurrency, and type abstraction.

In this organization, the chapters that describe the basic facilities make forward references to the existence of the more advanced ones. For example, Chapter 3 on types describes the Gypsy type mechanism and mentions all of the possible types. However, only the simple types and the static type compositions are described there. The others are described in later chapters. The index of this report gives the page that defines each phrase in the language.

This report on Gypsy 2.05 immediately supersedes the report on Gypsy 2.0. There will, however, be a period of transition during which the Gypsy Verification Environment (GVE) will continue to operate on Gypsy 2.0. Because Gypsy 2.05 consists mainly of a large subset of Gypsy 2.0, this report on Gypsy 2.05, for the greatest part, is also a report on Gypsy 2.0. The cases where Gypsy 2.05 differs from Gypsy 2.0 are described in a separate document, entitled "Differences in Gypsy Dialects," by Lawrence M. Smith and Robert L. Akers. During the transition period, these two documents together may serve as a report on Gypsy 2.0.

Acknowledgements

The contributions of Allen L. Ambler, Robert L. Akers, Richard E. Alterman, William R. Bevier, Woodrow W. Bledsoe, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Carol A. David, Benedetto L. DiVito, Dwight F. Hare, Charles G. Hoch, Gary R. Horn, John H. Howard, James C. Hsu, Lawrence W. Hunter, James Keeton-Williams, John McHugh, Judith S. Merriam, Mark S. Moriconi, Karl Nyberg, Ann E. Siebert, Lawrence M. Smith, Michael K. Smith, Russell A. Still, Anand V. R. Tripathi, Robert E. Wells and William D. Young to the development, implementation and initial experimental applications of Gypsy are gratefully acknowledged. Special acknowledgment is given to Robert L. Akers who prepared much of the material in the appendices of this report.

Pascal was the starting point for the development of Gypsy, and there are still strong semantic similarities. The languages Algol 60, Algol 68, Alphard, CLU, Concurrent Pascal, Euclid, Fortran, Nucleus, Simula and Special and the structured programming principles pioneered by Edsger W. Dijkstra and C. A. R. Hoare also have provided an assortment of fruitful ideas from which to draw.

The development, implementation and initial experimental applications of Gypsy have been sponsored primarily by the National Computer Security Center (Contracts MDA904-80-C-0481, MDA904-82-C-0445). Additional sponsorship has been provided by the U. S. Space and Naval Warfare Systems Command (formerly Naval Electronic Systems Command) (Contract N00039-81-C-0074), by the U. S. Air Force Rome Air Development Center (Contract F30602-84-C-0081), by Digital Equipment Corporation, by Digicomp Research Corporation, and by the National Science Foundation (Grant MCS-22039).

Chapter 1
BASIC CONCEPTS

Gypsy is a language for specifying, implementing and proving computer programs. A specification describes *what* effect is desired when a program runs, an implementation defines *how* the effect is caused, and a proof *verifies* that the program always runs as specified.

1.1 Programs

A Gypsy *program* is a mechanism whose operation causes an effect on its environment. The *environment* of a Gypsy program consists of *data objects* and *exception conditions*. Every data object has a name and a value. The *only* ways that running a program can cause an effect on its environment are by changing the value of a data object or by signalling a condition. A data object always has some value specified by the *type* of the object. Normally, a program causes an effect on its environment by changing the value of some data object. A program also, however, can *signal* an exception condition. Normally, this is done only to indicate that something unusual has happened.

1.2 Specification

A Gypsy *specification* is a declarative statement about the environment of a program. Every program *must* have a environment specification, and it also *may* have operational specifications.

The *environment specification* names *every* data object and exception condition in the environment. It also specifies the type of each data object and whether the object is variable or constant. The program can change the value of a *variable object*, but it can not change the value of a *constant object*. The environment specification completely isolates the effects of running the program. The only objects that a program can have access to are those that are named in its environment specification, and the only ones that it can change are its variable objects.

The environment specification provides a very weak, but complete, description of the effect caused by running a program. It defines completely what objects are in the environment and it identifies all those that *might* be changed as a result of running the program. Operational specifications may be used to state much stronger specifications. An *operational specification* gives a statement about what values the data objects may have as the program runs. An operational specification may make a very strong statement that describes many properties about the effect that is to be caused by a program, or it may make only a very weak statement that describes only a few properties. The strength of a specification is matter of human choice.

1.3 Implementation

A Gypsy *implementation* of a program is an imperative statement of how the program causes its effect. A program is implemented by describing how it is composed of pre-defined programs. Some of these are *standard* programs that are pre-defined by the Gypsy language, and others may be pre-defined by a particular implementation of the language (for example to provide i/o support on a particular machine).

1.4 Proof

Gypsy supports proofs about specifications and proofs about programs. Specifications are stated in terms of compositions of mathematical functions. Theorems about these functions and their compositions can be stated directly in the Gypsy specification language, and they can be proved in the Gypsy proof system.

The specifications of a program define constraints on its implementation. The specifications of a program can be viewed as sensors that are attached to its environment. The specification sensors are applied to the environment at various times as the program runs. Whenever a specification sensor is applied, its gives a value of either true or false. The implementation of a program *satisfies* its specifications if and only if all of its specification sensors give true *whenever* the program runs.

The main distinguishing characteristic of Gypsy is that is possible to give formal, mathematical proofs that a program satisfies its specifications. Gypsy is designed so that it always is possible to construct a set of logical formulas, called *verification conditions*, that are sufficient (but not always necessary) to show that the implementation of a program satisfies its specifications. If these formulas can be proved, then, whenever the program runs, its implementation causes an effect that satisfies its specifications.

The main reason for proving that an implementation satisfies its specifications is to give a sound, objective, convincing argument that the program is reliable -- that it always does what a user expects of it. It is tempting to believe that a proved program is totally reliable -- that it never can produce an unexpected result. This, however, provides a false sense of safety because there are several reasons why even a proved program sometimes may not run as expected. First, the selection of specifications often is quite subjective. Specifications may make a strong statement about what effect is expected, or they may make only a weak statement, or they may even say something that is not expected at all! In general, there is no objective way to determine if the specifications require that the program do exactly everything that a user will expect of it. Second, every proof is based on certain assumptions. If these assumptions are not valid, then the conclusions drawn from the proof may not be valid either. Third, any mechanical tools that are used to help construct a proof must produce a valid one. Fourth, a proof of a Gypsy program assumes that the combination of the supporting software and hardware, which implement the Gypsy language, satisfy exactly the Gypsy semantics. If any of these assumptions are violated, running the program may cause effects that do not satisfy its specification. A proved program normally is more reliable than an unproved one; but, one must understand clearly the specifications that are stated and the assumptions upon which the proofs are based.

1.5 Independence Principle

Gypsy is designed so that the proof of a program requires only certain, limited specifications about its components. An implementation of its components, for example, is never required. This characteristic is known as the *independence principle*, and it has two very important consequences. First, a large, complex program can be proved component by component in small, manageable steps. Second, the proof of a program can be done *before* its components are implemented. The word `PENDING` can be used in many places in Gypsy to indicate components that will be supplied in subsequent stages of development.

Because of the independence principle, Gypsy can be used throughout all the normal stages of the software life cycle. For example, at the highest level of system design, specifications can be stated for the total system. Then, the Gypsy program composition rules can be used to state how the system is implemented by its subsystems, and specifications can be given for these. Even at this very early stage, a proof of the total system can be constructed. Of course, this proof is contingent upon the subsystems satisfying their specifications, and consequently, a subsequent step must be to prove the subsystems in a similar way. This process can be applied repeatedly until all system components are decomposed into the pre-defined Gypsy programs. In this way, the Gypsy specification, implementation and proof methods can be applied throughout all levels of system composition, from the highest level of system design to the lowest level of coding.

1.6 Language Summary

The specifications and implementations of all Gypsy programs are written in terms of the five kinds of Gypsy units: `PROCEDURE`, `FUNCTION`, `CONST` (constant), `LEMMA` and `TYPE`. The fundamental units are types and procedures. A type specifies constraints on data objects. All Gypsy programs are procedures. A function is a special kind of procedure, and constants and lemmas are special kinds of functions. By providing pre-defined units and rules for composing units into user-defined units, Gypsy provides a program implementation language that includes data assignment, condition handling, dynamic memory (without explicit pointers) and concurrency. It also provides a specification language, including type abstraction, for stating desired properties of these implementations.

The Gypsy standard types are `BOOLEAN`, `CHARACTER`, `INTEGER`, `RATIONAL`, and `ACTIVATIONID`. Scalar types also may be defined. The standard type compositions are `ARRAY`, `RECORD`, `SET`, `SEQUENCE`, `MAPPING`, and `BUFFER`. Standard functions are pre-defined on all of these. Integer, boolean, rational, set, sequence and mapping are the familiar structures from ordinary mathematics, and the other types also have precise mathematical definitions. In so far as possible, the Gypsy standard functions are the ones normally associated with the mathematical structures. The well developed properties of these structures and their functions provide much of the power of the Gypsy proof methods. Logical deductions can be made about the Gypsy structures in the same way as the mathematical structures because, in most cases, the Gypsy structures *are* the mathematical structures.

The Gypsy standard procedures are assignment (`:=`), `NEW`, `MOVE`, `REMOVE`, `SEND`, `RECEIVE` and `GIVE`. New, move and remove are standard procedures for dynamic memory management. Send, receive and give are standard procedures for handling buffers. `BUFFER` objects are the only objects in Gypsy that can be shared among procedures that run concurrently. The sequential procedure compositions are `IF`, `CASE`, `LOOP` and `BEGIN`; and the concurrent procedure compositions are `AWAIT` and `COBEGIN`.

`ENTRY`, `BLOCK` and `EXIT` statements are operational specifications about the effect of a program on its external environment. `KEEP` and `ASSERT` statements are internal operational specifications. A relation among functions can be specified by a lemma. `CENTRY`, `CBLOCK`, `CEXIT` and `HOLD` are specification statements for type abstraction.

1.7 Language Implementation

Many of the Gypsy types are potentially unbounded in size, and therefore, it is impossible to make all Gypsy programs run on a real machine with finite resources. All of the Gypsy specification, implementation and proof methods are perfectly sound for these objects of unbounded size, but obviously, there always can be some value that is too big to fit in the available storage capacity of a real machine.

Therefore, every implementation of Gypsy for a real machine must restrict the programs that can run on that machine to some subset of Gypsy. There are no restrictions on how this subset may be chosen. The implementation for a particular machine also may provide an implementation prelude. An *implementation prelude* simply pre-defines a set of Gypsy units in addition to the standard ones. Defining a Gypsy subset and an implementation prelude can be used to tailor Gypsy to the individual characteristics of any particular machine. Clearly, this constrains the portability of Gypsy programs. However, if a Gypsy program can be ported from one machine to another, then so can its proof.

1.8 Verification Environment

Gypsy is designed to be implemented within an integrated programming environment. This programming environment is an interactive system that is intended to provide the all the tools needed to specify, implement and prove Gypsy programs throughout their life cycle. Because this environment needs to include tools for verifying programs, as well as other more conventional tools such as compilers, it is referred to as a *Gypsy Verification Environment* (GVE). A GVE consists of two major parts, a data base and a set of tools for working on information in the data base. The data base serves as a library for Gypsy units and other supporting information. The Gypsy units in the library may be in various stages of evolution. As the library evolves, new units may be added and old ones may be modified or deleted. The library may contain units from one or more computing systems, and the same units may be used in several systems. The library also may contain other supporting information such as verification conditions and their proofs. Ideally, a GVE contains *all* relevant information about the Gypsy units it contains (including even such things as documentation), and it contains mechanisms for ensuring the consistency of this information. There are no specific requirements on what tools must be provided by a GVE. Ideally, it should provide all the tools that a program developer needs to support the evolution of a formally specified and proved program throughout its life cycle. Tools are needed to create and modify Gypsy text, to prove the programs, to make them run and to produce whatever other kinds of information can be contained in the library. The basic tools that are needed are a text editor, verification condition generator, theorem prover and compiler. In fact, it is quite reasonable for one GVE to contain several different compilers for several different target machines. These compilers may be somewhat different from conventional compilers because they compile from units in the library rather than directly from a text file.

Chapter 2
LEXICAL PRELIMINARIES

Every segment of text in the Gypsy language is a sequence of ASCII characters. The characters are grouped into words which form the five kinds of Gypsy *units*: procedures, functions, constants, lemmas, types. Finally, Gypsy units are organized into scopes.

2.1 Notation

The phrases of characters and words that may appear in Gypsy text are identified by English words embedded in <...>. These same words are used to describe the meaning of the phrase. Each phrase is defined in terms of other phrases, words and characters. For example,

```
<identifier> ::= <letter> { [ _ ] <letter or digit> }
```

The symbol `::=` means "is defined as." In defining the phrases, | means "or", parts enclosed in [...] are optional and parts enclosed in {...} may appear zero or more times. For example, the preceding defines an identifier as a sequence of letters, digits and underscores that begins with a letter and does not end with an underscore.

In defining phrases, any letter or word that appears literally as a part of the phrase is given in upper case. Also, unless stated otherwise, there may be any number of space, tab, carriage return or line feed characters between phrases.

2.2 Character Set and Conventions

Gypsy text is composed of ASCII characters. Unless specified otherwise, upper and lower case letters can be used interchangeably, and matching pairs of parentheses (...) and square brackets [...] may be used interchangeably. However, in this report, to distinguish Gypsy text from the text of the report, Gypsy text will be written only in upper case. Also, Gypsy text will contain only (...) so that [...] can denote optional parts of phrases.

2.3 Identifiers

Identifiers are used as names for Gypsy units and objects.

```
<identifier> ::= <letter> { [ _ ] <letter or digit> }

<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

  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The sequence of characters of an identifier must be contiguous, and any character immediately either before or after an identifier must be some character other than a letter, digit, or underscore. The identifiers given in Appendix A have a pre-defined meaning, and they can not be used for any other purpose.

Examples: X Y Z4 WIDTH AREA_OF_BOX

2.4 Comments

Comments are annotations that are embedded in the Gypsy text. These annotations have no effect on the meaning of the text. If they are removed from the text, the meaning of the text is not changed. Any number of comments can be placed before or after any Gypsy phrase.

```
  <comment> ::= "{" { <comment character> } "}"

  <comment character> ::= any character except "}"
```

The symbols "{" and "}" above mean literally the characters { and }.

```
Examples:  {VARIABLE X CORRESPONDS TO THE CRT}
           {THIS PROCEDURE LAST MODIFIED ON MARCH 1, 1983}
```

Chapter 3
TYPE SPECIFICATIONS

A Gypsy *type* specifies constraints for data objects. A type specification must be given for every data object. A type specification defines a set of values, and a data object always must have some value in the set of values defined by its type. Every type specification also defines a default initial value, and some type specifications may specify certain additional constraints. There are only two ways in which a type may be defined. It may be a pre-defined type, or it may be some composition of existing types. Thus, every type is either pre-defined or it is some composition of the pre-defined types. There are a fixed set of rules for composing types. An object that has a composed type is called a *structured object* (because it is a structure that has several components).

```
<type specification> ::= <type name> | <subrange type>
                         | <restricted buffer type composition>

<type name> ::= <identifier>
```

A type name may be the name of a pre-defined type, or it may be the name of a type that is defined by a type declaration.

```
<type declaration> ::= <ordinary type declaration>
                     | <abstract type declaration>

<ordinary type declaration> ::=
  TYPE <type name> = <type definition>

<type definition> ::=
    PENDING | <scalar type> | <static type composition>
  | <dynamic type composition> | <buffer type composition>
  | <type specification> [ := <default initial value expression> ]
```

`PENDING` is a place holder for some unknown type definition.

```
Example type declaration: TYPE INT = INTEGER
```

3.1 Default Initial Values

Every type has a *default initial value* that is assigned to an object of that type when it is created (unless specified otherwise). The default initial value ensures that every data object *always* has some value of its specified type. A data object never has an undefined value. The standard function `INITIAL` gives the default initial value of a type.

```
<default initial value expression> ::= <pre-computable expression>
```

The use of a default initial value expression in a type definition defines a type that is the same as the one given

by the type specification except that the default initial value is the one given by the initial value expression. The default initial value must satisfy the type specification that it follows.

```
Example type declaration with default initial value expression:
  TYPE INT2 = INTEGER := 2
  {INT2 is the same type as INTEGER except that its default
   initial value is 2 (instead of 0)}
```

3.2 Simple Types

The simple standard types are boolean, character, integer, rational. Scalar types and the subrange types also may be defined.

3.2.1 Scalar Types

A scalar type defines a non-empty sequence of scalar values. Its default initial value is the first value in the sequence.

```
<scalar type> ::= ( <scalar value> { , <scalar value> } )

<scalar value> ::= <identifier>
```

A scalar type definition `TYPE t=(v0, ..., vn)` defines a set of Gypsy constants (Section 5.5) `CONST v0:t = 0; ...; CONST vn:t = n`. The integer values of these constants can be obtained *only* by using the standard function `ORD` -- for example, `ORD(v0) = 0, ..., ORD(vn) = n`.

```
Example type declaration of a scalar type:
  TYPE COLOR = (RED, BLUE, GREEN)
  {This defines CONST RED:COLOR=0 ... CONST GREEN:COLOR=2
   and ORD(RED)=0, ..., ORD(GREEN)=2.}
```

3.2.2 Type Boolean

Type `BOOLEAN` is the standard scalar type of logical truth values.

```
TYPE BOOLEAN = (FALSE, TRUE)
```

3.2.3 Type Character

Type `CHARACTER` is the standard type that defines scalar values for the 128 member ASCII character set. The scalar values for the printable ASCII characters " " (space) through "~" are named by non-standard "identifiers" consisting of a character between two single quote marks. For example, `'x'` names the lower-case character x. These "identifiers" can be used in the same way as normal identifiers (that name a scalar value). The other ASCII characters do not have predefined names in Gypsy 2.05, so they must be constructed with the standard function `scale`.

3.2.4 Type Integer

Type `INTEGER` is the set of whole numbers of ordinary mathematics. The default initial value is `0`.

```
<integer value> ::= [ - ] <number>

<number> ::= [ <base> ] <digit> { <digit> }
```

```
<base> ::= BINARY | OCTAL | DECIMAL | HEX
```

The digits in a number must be contiguous, and they must be of the base indicated. If no base is given, `DECIMAL` is assumed. Any character either immediately before the first digit or immediately after the last digit must be some character other than a digit.

```
Examples:  17    -3    BINARY 101111    - HEX 3F9
```

3.2.5 Type Rational

Type `RATIONAL` is the set of rational numbers of ordinary mathematics. The default initial value is `0/1`.

```
<rational value> ::= <integer value> / <number>
```

Note that the bases of the numerator and the denominator are stated *separately* -- for example, `OCTAL 777/111` is the rational number `(OCTAL 777)/(DECIMAL 111)`.

```
Examples:  2/4    -17/2
```

3.2.6 Subrange Types

A *subrange* type is a simple type with a value set that is restricted to a limited range.

```
<subrange type> ::= <simple type name> <range restriction>

  <range restriction> ::= <non-empty pre-computable range>

<simple type name> ::= <type name>

  <non-empty pre-computable range> ::= <range>

<range> ::= ( <range limits> )

<range limits> ::= <minimum value> .. <maximum value>

<minimum value> ::= <expression>

<maximum value> ::= <expression>
```

Both the minimum and maximum values in the range restriction must be pre-computable (Section 4.3) values of the simple type named, and the minimum must be less than or equal to the maximum.

The subrange type is the same as the simple type named except for the following. The value set specified by the subrange type is set of values from the minimum to the maximum value or the range limits. If the default initial value of the simple type named is not in the range limits, the default is the minimum value in the range.

```
Examples of type declarations of subrange types:
 TYPE OCTAL_INT = INTEGER(0..7)
  TYPE PRINTABLE_CHAR = CHARACTER('!'..'~')
  TYPE DIGIT = PRINTABLE_CHAR('0'..'9')
  TYPE WORK_DAY = DAY(MONDAY..FRIDAY)
  TYPE BASE_INTERVAL = RATIONAL(1/2..3/2)
```

3.3 Static Type Compositions

The static type compositions compose values that have a fixed number of components.

```
<static type composition> ::= <array type> | <record type>
```

3.3.1 Arrays

An *array* has a fixed number of components each of the same type. Each component is called an *element*. The default initial value of an array is the value obtained by assigning the default initial value of the element type to each element.

```
<array type> ::= ARRAY ( <index type> ) OF <component type>

<index type> ::= <non-rational simple type specification>

<non-rational simple type specification> ::= <type specification>

<component type> ::= <type specification>
```

The index type may be any simple type except rational or a subrange of rational.

An array has one element for each value of its index type. Each element has a selector *i*, which is an element of the index type, and a value *v* of the element type. The value of an array is *array*:{(*i1*,*v1*), ..., (*in*,*vn*)}. The *{...}* part is the set of components, and the tag *array* marks the set as the value of an array.

```
Examples of type declarations of array types:
  TYPE INT_ARRAY = ARRAY (INTEGER(1..10)) OF OBJECT
  TYPE OBJECT = ARRAY (CHARACTER) OF BOOLEAN
```

3.3.2 Records

A *record* has a fixed number of components which may be of different types. Each component of a record is called a *field*. The default initial value of a record is the value obtained by assigning each of its fields the default initial value of its type.

```
<record type> ::= RECORD ( <fields> )

<fields> ::= <similar fields> { ; <similar fields> }

<similar fields> ::= <field name> { , <field name> } : <field type>

<field name> ::= <identifier>

<field type> ::= <type specification>
```

Each field has a name *f* and a value *v* which is of the type of that field. The name of each field must be unique within the record. The value of a record is *record*:{(*f1*,*v1*), ..., (*fn*,*vn*)}. The value of a record is a *set* marked with the tag *record*. Therefore, the order in which its fields are defined is unimportant.

```
Examples of type declarations of record types:
  TYPE DATE = RECORD(MONTH: MONTH_ID;
                     DAY: INTEGER(1..31);
                     YEAR: INTEGER(1900..2000))
```

3.4 Base Types

Base types are used to determine if data objects are suitable operands for Gypsy programs. In general, the *base type* of a type is the type that is composed in the same way but with its various restrictions removed. The precise definitions of the base types are given in Appendix B.

Chapter 4
EXPRESSIONS

Expressions are rules for computing values. There are two rules for computing values. A value may be computed by i) taking the value of a data object, or ii) by computing a function of other values. In some contexts, expressions also are rules for computing names of data objects. (The conditions that can be signalled while computing expressions are described in Section 8.5.4 and Appendix C.)

4.1 Name Expressions

A *name expression* is a rule for computing the name of a data object. If the object is a structured object, each of its components is also an object and a name expression may name either the entire structured object or one of its components.

```
<name expression> ::= <data object name> { <component selectors> }

<data object name> ::= <identifier>
```

The data object name must be the name of some existing data object. If the data object name is followed by component selectors, it must be a structured object and the name expression names the selected component.

Examples: X R.F Y(I) Z(I)(J) Z(I,J) S.F(I,J).G

4.1.1 Component Selectors

Component selectors identify some one component or value (of one component) of a structured object.

```
<component selectors> ::=
    . <field name>
  | ( <index selector> { , <index selector> } )


<index selector> ::= <expression>
```

The component selectors are applied in order from left to right, and the form (*i*,*j*,...) is an abbreviation for the selector sequence (*i*)(*j*).... A field name is a selector for record fields, and an index selector is a selector for arrays, sequences, and mappings. Except in some cases for mapping components (Section 9.3.1), an index selector must name an existing component; otherwise, a condition is signalled.

4.2 Value Expressions

A *value expression* may be a literal value or the value of a data object, or it may be the result of applying various functions to other values. Some of these functions are denoted by special operators.

4.2.1 Primary Values

The basic parts of a value expression are its *primary values*.

```
<primary value> ::= <literal value> | <set or sequence value>
                  | <entry value>   | <data object name>
                  | <function call>

<literal value> ::= <scalar value>    | <integer value> | <character value>
                  | <rational value>  | <string value>
```

If the primary value is a data object name, its value is the value of the data object it names.

4.2.2 Modified Primary Values

If a primary value is the value of a structured object, its component values may be selected or they may be altered.

```
<modified primary value> ::= <primary value> { <value modifiers> }

<value modifiers> ::= <value selectors> | <value alterations>

<value selectors> ::= <component selectors> | <subsequence selector>
```

The value modifiers are applied in order from left to right, and the component selectors are used in the same way as in name expressions.

4.2.3 Value Alterations

Value alterations define a value of a composed type. The value is defined in terms of another similarly composed value with the values of some of its components altered.

```
<value alterations> ::=
    WITH ( <component alterations> { ; <component alterations> } )

<component alterations> ::= [ <each clause> ] <component modification>

<component modification> ::=
    <component assignment> | <component creation> | <component deletion>

<component assignment> ::= <alteration selector list> := <expression>

<alteration selector list> ::=
    <component selectors> { <component selectors> }
```

Each of the component alterations modifies one or more component values of a composed type. The modifications are done *in sequence* from left to right. A component assignment assigns the value of its expression to the component named by its alteration selector list. If the value is not within the type of the component, a condition is signalled.

Examples of expressions with value alterations:

```
{ARRAY} A WITH ( (I) := X )
          {This value, say U, is the same as array A except U(I)=X.
           For all other elements, U(I)=A(I).}

{RECORD} R WITH ( .F := Y; .G(K) := Z)
          {This value, say V, is the same as record R except that V.F=Y
           and V.G(K)=Z.  The other components of V and R are equal.}

{STRUCTURE} S WITH ( .F(I,J).G := Z )
          {This value, say W, is the same as structure S except that
           W.F(I,J).G=Z.  The other components of W and S are equal.}
```

4.2.4 Each Clauses

An *each clause* designates a sequence of operations.

```
<each clause> ::= EACH <identifier> : <bounded index type> ,
<bounded index type> ::= <index type>
```

Within value alterations, an each clause has the form

```
EACH <identifier> : <bounded index type> , <component modification>
```

This designates a sequence of component modifications. The index type, which must have a smallest and a largest value, defines an ordered sequence of simple values. These values are bound successively, from smallest to largest, to the identifier. This defines a sequence of component modifications (one for each value of the index type) which are performed in order.

The appearance of the identifier in the each clause defines it as a local name (Section 5.3) of the Gypsy unit that contains the each clause. The identifier of an each clause may be used only within the component modification of the each clause (as a local constant of its index type). Within the component modification, the identifier may not be used as the bound identifier in another each clause or quantified expression (Section 6.1.2).

```
Example expression with each clauses:
  {ARRAY} A WITH ( EACH I : SMALL_INT, (I) := F(I) )
            {If TYPE SMALL_INT=INTEGER(1..4), then this is the same value as
             A WITH ( (1):=F(1); (2):=F(2); (3):=F(3); (4):=F(4) ).}
```

4.2.5 Operators

A value of an expression may be simply a modified primary value, or it may be computed by applying operators to modified primary values as operands. *Operators* in Gypsy are a special notation for calls of standard functions.

```
<expression> ::= <term> | <quantified factor>

<term> ::= <factor> { <binary operator> <factor> }
                    [ <binary operator> <quantified factor> ]

<factor> ::= [ <unary operator> ] <modified primary value>
           | <if expression>
           | ( <expression> )

<unary operator> ::= <integer unary operator>
                   | <rational unary operator>
                   | <boolean unary operator>
```

```
<binary operator> ::= <simple relational operator> | <boolean operator>
                    | <integer operator> | <rational operator>
                    | <set operator>     | <sequence operator>
                    | <mapping operator>

<simple relational operator> ::= EQ | = | NE | < | LT | LE | > | GT | GE

<boolean unary operator> ::=  NOT

<boolean operator> ::= AND | & | OR | IMP | -> | IFF

<integer unary operator> ::=  -

<integer operator> ::= ** | * | / | DIV | MOD | + | -

<rational unary operator> ::=  -

<rational operator> ::= ** | * | / | + | -

<quantified factor> ::= [ <boolean unary operator> ] <quantified expression>
```

If an expression has no operators, its value is its modified primary value. If it does have operators, its value is the result of applying the operators to their operands.

The operand that an operator is applied to is determined by precedence levels. Operators with lower numbered precedence levels are performed first; and among operators of equal precedence, the operators are applied from left to right. (The one exception is the `:>` operator. The operations *x* `:>` *y* `:>` *z* `:>` *s* are performed from right to left.) The precedence levels are as follows:

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

The operands in an expression must satisfy the type requirements of the standard function denoted by their operator. These type requirements and the results produced by the operators are given in Appendix C.

```
Examples of expressions:   X       A(I)       X.F < M(I,J) + 1 AND B
```

4.2.6 If Expression

An *if expression* provides a way of choosing one of several potential values as the value of an expression.

```
<if expression> ::=
    IF <boolean expression> THEN <potential value expression>
    { ELIF <boolean expression> THEN <potential value expression> }
    ELSE <potential value expression> FI

<potential value expression> ::= <expression>
```

The value of the if expression is the value of the potential value expression that follows the first boolean expression that is true. If none of them are true, the value of the if expression is the value of the last potential value expression. The only boolean expressions whose values are computed are those needed to determine the first one that is true. The only potential value expression whose value is computed is the one that defines the value of the if expression.

Example: `IF X < Y THEN P+Q ELSE 4 FI`

```
IF   CH < A THEN "LESS"
ELIF CH = A THEN "EQUAL-LESSER"
ELIF CH < B THEN "BETWEEN"
ELIF CH = B THEN "EQUAL-GREATER"
             ELSE "GREATER" FI
```

4.3 Pre-Computable Expressions

A *pre-computable expression* is one whose value can be computed in a particular Gypsy verification environment prior to running a Gypsy program on a target machine. Thus, what is pre-computable in Gypsy may vary from one environment to another.

```
<pre-computable expression> ::= <expression>

<pre-computable value> ::= <scalar value> | <integer value>
                         | <string value> | <constant name>
                         | <type name>
```

A pre-computable expression is one that is composed only of pre-computable values, standard functions and operators. Type names are pre-computable values only insofar as they may appear as actual parameters to the standard functions UPPER, LOWER, SCALE, INITIAL, and NULL. Each Gypsy verification environment determines what set of values, standard functions and operators are pre-computable in that environment.

Examples of expressions that *may* be pre-computable:

```
14
TRUE
M+N {Provided M and N are constant units}
SCALE(1,COLOR)
INITIAL(NUM_ARRAY) WITH ( EACH I:INDEX, (I) := I )
```

Chapter 5
PROGRAMS

A Gypsy program can cause an effect on its environment *only* by changing the value of its data objects or by signalling a condition (Chapter 8). Complete external environment specifications *must* be given for every program. Operational specifications, an internal environment, an implementation and internal specifications also *may* be given. All Gypsy programs are procedures. Functions are a special kind of procedure, and constants are a special kind of function.

5.1 Procedures

A Gypsy *procedure* is a mechanism that causes some effect on an environment of data objects and conditions.

```
<procedure declaration> ::=
    PROCEDURE <procedure name>
     <external data objects> [ <external conditions> ] =
    <procedure body>

<procedure name> ::= <identifier>
```

The following is an example of a complete Gypsy procedure with its supporting type and constant declarations. (All Gypsy units must be contained in scopes. See Chapter 7.)

```
SCOPE MATRIX =
BEGIN

  PROCEDURE COLUMN_SUM(VAR S:A_LARGE_INT; A:A_MATRIX; COLUMN:AN_INDEX)=
  BEGIN
    VAR I:AN_INDEX := 1;
    S := 0;
    LOOP
      S := S + A(I,COLUMN);
      IF I = MATRIX_SIZE THEN LEAVE {THE LOOP}
      ELSE I := I + 1;
      END;
    END;
  END;

  CONST MATRIX_SIZE:INTEGER = 10;

  TYPE AN_INDEX = INTEGER(1..MATRIX_SIZE);
  TYPE A_MATRIX = ARRAY (AN_INDEX) OF AN_ARRAY;
  TYPE AN_ARRAY = ARRAY (AN_INDEX) OF A_SMALL_INT;
```

```
    CONST MAX_SMALL_INT:INTEGER = 1000;
    TYPE A_SMALL_INT = INTEGER(-MAX_SMALL_INT..MAX_SMALL_INT);

    CONST MAX_LARGE_INT:INTEGER = 10 * MAX_SMALL_INT;
    TYPE A_LARGE_INT = INTEGER(-MAX_LARGE_INT..MAX_LARGE_INT);

  END;
```

5.2 External Environment

An *external object* is one whose life time extends beyond the interval during which the procedure runs. The external data objects and conditions of the procedure define its complete *external environment*. It has access to *no* other external objects. The external data objects and conditions define the *formal parameters* of the procedure. A *formal parameter* is a name that is used temporarily, while the procedure runs, to refer to an external object. (It does, however, have access to other Gypsy units (Section 7.4). Gypsy units are neither data nor condition objects.)

```
    <external data objects> ::=
        ( <similar formal data parameters>
          { ; <similar formal data parameters> } )

    <similar formal data parameters> ::=
        [ <access specification> ] <formal data parameters> : <formal type>

    <access specification> ::= VAR | CONST

    <formal data parameters> ::= <identifier> { , <identifier> }

    <formal type> ::= <type specification>
```

The formal parameters are local names of the procedure. (See Section 5.3.) Every formal data parameter has an access specification and a formal type specification. An access specification of `VAR` specifies that the external object is a variable object, and `CONST` specifies that it is a constant object. If no access specification is given, `CONST` is assumed. The procedure may assign a value to a variable object, but not to a constant object. Within the procedure, each object must have a value of its formal type.

5.3 Local Names

Every Gypsy unit has associated with it a set of local names. The *local names* are identifiers which are contained within the unit and which identify various parts of the unit. The local names include the following: the unit name, the names of the formal parameters including `MYID` (Section 10.6.1) and `RESULT` (Section 5.4), the names of the internal data objects and conditions, the names of all external units referred to within the unit. Each of the local names mentioned above must be unique within the unit.

Quantified names in quantified expressions (Section 6.1.2) and the names of identifiers in each clauses (Section 4.2.4) are the *bound identifiers* of the unit. Each bound identifier is also a local name and it must be different from each of the local names listed in the preceding paragraph. Bound identifiers need not be unique within a unit so long as they satisfy the restrictions stated in Sections 6.1.2 and 4.2.4.

5.4 Functions

A Gypsy *function* is a procedure that has exactly one formal variable parameter named RESULT. RESULT is the *value* of the function.

```
<function declaration> ::=
    FUNCTION <function name> [ <equality extension> ]
             [ <external data objects> ] : <result type>
             [ <external conditions> ] =
             <procedure body>

<function name> ::= <identifier>

<result type> ::= <type specification>
```

The external data objects, external conditions and body are the same as for procedures *except* that a function may have *only* constant formal data parameters. Therefore, running a function can not produce any effect on its external data objects except to give a value to its RESULT.

The RESULT parameter for a function is specified automatically and implicitly. It may *not* appear explicitly in the list of formal parameters, but it may be used throughout the rest of the function just like an ordinary formal variable parameter. The result type is the formal type of the RESULT parameter. (When a function is called, an object whose value is the default initial value of the result type is used as the actual parameter for RESULT.)

The Gypsy proof methods assume that all functions are deterministic -- that is, if given the same values for its constant formal parameters, the function always produces the same value of RESULT. Determinism is not assumed for procedures. Most functions that can be written in Gypsy are deterministic, but there are a few cases in which (by virtue of concurrency and type abstraction) it is possible to write non-deterministic ones. Thus, for a Gypsy proof to be valid, every function that it refers to must be proved to be deterministic.

```
Example:  FUNCTION CSUM(A:A_MATRIX; COLUMN:AN_INDEX) : A_LARGE_INT =
          BEGIN
            COLUMN_SUM(RESULT,A,COLUMN);
          END;
```

5.5 Constants

A Gypsy *constant* is a concise notation for a pre-computable function with no constant formal parameters.

```
<constant declaration> ::=
    CONST <constant name> : <result type>  := <constant body>

<constant name> ::= <identifier>

  <constant body> ::= PENDING | <pre-computable expression>
```

The constant body must produce a value of the result type. The definitions of constants may not be circular. PENDING is a place holder for some unknown constant body.

```
Examples:  CONST N : INTEGER := 14
           CONST M : A_COLOR := SCALE(1,A_COLOR)
           CONST LAST_DAY : A_DAY := PENDING
           CONST INITIAL_VECTOR : A_VECTOR
                 := INITIAL(A_VECTOR) WITH ( EACH I:AN_INDEX, (I) := 2*I )
```

5.6 Bodies

A *procedure body* defines the external operational specifications, implementation and internal specifications of a procedure. The implementation may define an internal environment and it also defines the imperative statements that cause the effect of the procedure. The internal environment is a set of objects that exist only while the procedure runs. The external and the internal environment of a procedure are its total environment. It has access to no other data or condition objects.

```
<procedure body> ::= PENDING
                   | BEGIN
                     [ <external operational specifications> ]
                     [ <internal environment> ]
                     [ <keep specification> ]
                     [ <internal statements> ]
                     END
```

The internal environment defines the internal objects of the procedure body, and the names of these objects are local names of the procedure. The internal objects may be referred to in all parts of the procedure body *except* in the external operational specifications. `PENDING` is a place holder for some unknown procedure body.

5.7 Internal Environment

The *internal environment* of a procedure consists of data objects and conditions that exist only while the procedure runs.

```
<internal environment> ::= <internal data or condition objects>
                           { <internal data or condition objects> }

<internal data or condition objects> ::= <internal data objects> ;
                                      | <internal condition objects> ;

<internal data objects> ::=
    <access specification>
    <internal data object names> : <type specification>
    [ := <internal initial value> ]

<internal data object names> ::= <data object name>
                           { , <data object name> }

<internal initial value> ::= <expression>
```

The internal data objects are created and initialized in order. If an internal initial value is not given, the default initial value of the type is the initial value of the internal data object. The only internal objects that can be referred to in an internal initial value expression are data objects created previously. If an internal initial value is not of the type of its internal object, a condition is signalled. See Section 8.5.3.) The access specification for an internal data object has the same meaning as for formal parameters.

```
Examples:  VAR I:INTEGER
           VAR PER_CENT : INTEGER := 100
           CONST SOUND_HYPOTHESIS : BOOLEAN := TRUE
           CONST IDENTITY_MATRIX : A_MATRIX := IDENTITY(N)
```

5.8 Internal Statements

The *internal statements* contain imperative statements about how the procedure causes its effect and they contain declarative statements of certain specifications. The imperative statements may be procedural statements (which are calls of standard procedures) or compositions of procedure calls. A procedure composition rule describes a way of calling one or more procedures. Therefore, a composition rule has the same general effect as calling a single procedure. It produces a state change in the objects of the environment. `PENDING` is a place holder for some unknown statement list.

```
<internal statements> ::= <statement list> [;] | PENDING [;]

<statement list> ::= <statement> {; <statement> }

<statement> ::= <procedural statement>   | <procedure composition rule>
              | <assert specification>

<procedural statement> ::=
            <assignment statement>
          | <give statement>
          | <leave statement>
          | <move statement>
          | <new statement>
          | <procedure statement>
          | <receive statement>
          | <remove statement>
          | <send statement>
          | <signal statement>

<procedure composition rule> ::=
    <if composition>    | <case composition>  | <loop composition>
  | <begin composition> | <await composition> | <cobegin composition>
```

5.8.1 Data Assignment

A *data assignment statement* is a call of the standard assignment procedure.

```
<assignment statement> ::=
    <variable name expression> := <expression>

<variable name expression> ::= <name expression>
```

The variable name expression must name some existing variable data object. The base type of the variable object must be the same as the base type of the value expression. The effect of the call is that the value of the variable becomes the value of the expression. A condition is signalled if the value is not of the type of the variable.

```
Examples:  X := Y
           A(I+J) := F(X)
           R.F(K) := IF G(Y) THEN X < 2 * P + 1 ELSE FALSE FI
```

5.8.2 Input and Output

Gypsy does not have any special statements for input and output. Input and output are done through whatever Gypsy objects are provided in the implementation prelude (Section 5.10.3). Input and output commonly are done with the send and receive statements on objects of type buffer as described in Chapter 10. However, input and output are defined solely by the implementation prelude, and it may provide whatever pre-defined objects and procedures are appropriate for a particular machine. Type string is sometimes defined by an implementation for input and output of string objects.

5.8.3 If Composition

An *if composition* chooses and performs one of several internal statement lists.

```
<if composition> ::=
    IF   <boolean expression> THEN [ <internal statements> ]
  { ELIF <boolean expression> THEN [ <internal statements> ] }
  [ ELSE [ <internal statements> ] ]
  [ <condition handlers> ]
    END
```

The internal statements are performed that follow the first boolean expression that is true. If none of the boolean expressions are true, the internal statements following the ELSE are preformed provided there is one; if not, none of the internal statements are performed. The only boolean expressions whose values are computed are those needed to determine the first one that is true.

```
Examples:  IF I = N THEN LEAVE
           END

           IF P(X,Y) THEN GENERATE(W)
           ELIF Q(X,Y) THEN RESTORE(W);
           ELSE LEAVE;
           END
```

5.8.4 Case Composition

A *case composition* is another way of choosing and performing one of several internal statements.

```
<case composition> ::=
    CASE <label expression>
  { IS <case labels> : [ <internal statements> ] }
  [ ELSE             : [ <internal statements> ] ]
  [ <condition handlers> ]
    END

<label expression> ::= <scalar or integer valued expression>

<scalar or integer valued expression> ::= <expression>

  <case labels> ::= <pre-computable label expression>
            { , <pre-computable label expression> }

<pre-computable label expression> ::= <integer value> | - <integer value>
                                    | <character value>
                                    | <scalar value> | <identifier>
```

Every case label must be unique and of the same base type as the label expression. The identifier in the case

label must represent a constant value. If the label expression is equal to one of the case labels in a branch of the case composition, the corresponding internal statements are performed. If there is no such case label, but there is an ELSE label, the internal statements after the ELSE are performed. If there is no such case label and no ELSE, the case composition does not perform any of its internal statement lists.

```
Example:  CASE X+1
          IS        2,7: MAKE_RED(Y);
          IS        - 9: MAKE_BLUE(Y);
          IS      THREE: MAKE_GREEN(Y);
          ELSE:          MAKE_BLACK(Y);
          END;
```

5.8.5 Loop Composition

A *loop composition* performs its internal statements repeatedly. A loop is terminated by either by performing a leave statement or by signalling a condition. Every leave statement must be contained within some loop statement. Performing a leave statement terminates its most tightly enclosing loop statement.

```
    <loop composition> ::=
        LOOP [ <internal statements> ] [ <condition handlers> ] END

    <leave statement> ::= LEAVE
```

```
Examples:  LOOP {FOREVER}
             GET_CHAR(C,SOURCE);
             PROCESS_CHAR(C,DATA_BASE);
           END

           LOOP
             COMPUTE_VECTOR(V,I);
             IF I=N THEN LEAVE {THE LOOP}
             ELSE I := I + 1;
             END;
           END
```

5.9 Procedure and Function Calls

A *procedure call* causes a procedure to run. This is most fundamental action in all of Gypsy. Running a procedure is the *only* way to cause an effect on an environment. The procedure that issues the call is the *calling* procedure. The procedure that runs on its behalf is the *called* procedure.

```
    <procedure statement> ::= <called procedure name> <actual parameters>

    <called procedure name> ::= <procedure name>

    <function call> ::= <called function name> <actual parameters>

    <called function name> ::= <function name>
```

The called procedure name must be one that is explicitly declared as a procedure. The standard procedures each are called with a special notation, and they may not be called with a procedure statement. Even though a function is regarded as a special kind of procedure, it may only be called with a function call.

```
Examples:  P(R.F,Y+Z)        G(T)        Q
```

5.9.1 Actual Parameters

The *actual parameters* of a program (procedure or function) call are the objects of the environment (of the calling program) that become the external objects of the called program. The called program has access to no other objects in the calling environment.

```
<actual parameters> ::=  [ <actual data parameters> ]
                       [ <actual condition parameters> ]

<actual data parameters> ::=
  ( <actual data object> { , <actual data object> } )

<actual data object> ::= <expression> | <variable name expression>
```

There must be one actual data parameter for each formal data parameter of the called program. A function may have zero or more data parameters, but a procedure must have at least one. While the called program runs, it uses the name of its formal parameter as a temporary name for the corresponding actual parameter. (This is normal call-by-reference parameter passage.)

If an actual data object corresponds to a variable formal parameter, it *must* be a variable name expression. The value set of the formal type of the parameter must be the same as, or a subset of, the value set of the type of its actual data object. (Without this requirement, running the program could cause the actual data object to have a value not allowed by its type.)

An actual data object that corresponds to a constant formal parameter may be any expression. If this expression is not a name expression, a uniquely named temporary object is created with the value of the expression, and the temporary is used as the actual object.

The following checks for type consistency and potentially harmful aliasing also are made (except as specifically noted in certain standard operations).

5.9.2 Type Consistency

All of the actual data parameters are checked to see if they meet their corresponding formal type specifications (as specified in the called program). The value of each actual data object must be in the value set of its corresponding formal type; otherwise, a condition is signalled.

5.9.3 Aliasing

A condition is signalled if there is any potentially harmful aliasing among the actual data parameters. *Aliasing* occurs if some actual data object can be referred to by more than one formal name. If so, these formal names are different aliases (names) for the same actual object. Aliasing is *potentially harmful* if changing the value of one formal variable parameter causes the value of another formal parameter to change. This is potentially harmful because the called program is specified and proved assuming that a change in the value of the external object referred to by its formal parameter does *not* cause a change in the value referred to by any other formal parameter.

5.9.4 Transfer of Control

When a program is called, the actual parameters are considered from left to right. First, the name or value of the actual parameter is computed, and then the value is checked for type consistency. After this has been done for each actual parameter, the checks for aliasing are made. If any of these conditions are signalled, control returns to the calling program, and these conditions are handled as described in Chapter 8. If none of these conditions are signalled, the called program starts running with its formal parameters serving as temporary names for the corresponding actual parameters. While the called program runs, the running of the calling program is suspended. When the called program stops running (if it does), the running of the calling program is resumed.

5.10 Getting Started

Normally, Gypsy programs are developed in a Gypsy Verification Environment (GVE), and the GVE provides the mechanisms for running Gypsy programs in a particular target environment.

5.10.1 Developing a Program

The normal way of starting a Gypsy verification environment is with a "GVE" command. The GVE should give various greeting information and pause with a prompt. The greeting information should instruct the user in how to get further information about the facilities available. Normally, this would be done by some kind of "HELP" or "?" command.

5.10.2 Running a Program

Any Gypsy program can be run within a target environment just by calling it with actual parameters that are defined within that environment. How the environment is designated and how this call is made are defined by the GVE, but the call must satisfy all of the normal Gypsy rules for a program call (Section 5.9). The data and condition objects of a target environment are defined by the GVE.

5.10.3 Implementation Prelude

An *implementation prelude* provides the connection between a Gypsy program and a run-time environment on a particular target machine. A prelude describes what Gypsy objects are available on a target machine to be passed as actual parameters to a Gypsy program. Running a Gypsy program on these actual parameters is how an effect is produced on the target machine. A user of Gypsy never will write an implementation prelude, but it often will be necessary to use one.

An implementation prelude always contains some target environment. A target environment defines Gypsy objects that the language implementor has provided on a target machine. These are the objects that can be used as the actual parameters to a main program. Normally, these are the objects that provide input and output facilities, but they may be whatever objects the implementor chooses to provide so long as they behave like normal Gypsy objects.

An implementation prelude also may include a set of Gypsy units that have been pre-defined by the language implementor (in addition to the standard Gypsy units). These units may be implemented in Gypsy, but they need not be. However, if not, their externally visible behavior *must* be as though they were normal Gypsy units. Regardless of how these additional pre-defined units are implemented, normal Gypsy specifications must be stated for them; and, in this way, proofs can be constructed for programs that use these units. These proofs are based on the assumption that the implementations of the pre-defined units satisfy the

specifications given.

Chapter 6
OPERATIONAL SPECIFICATIONS

The *operational specifications* of a program state constraints on its implementation. The specifications of a program can be viewed as sensors that are applied to the environment of a program at various times as it runs. Whenever a specification sensor is applied, its gives a value of either true or false. The implementation of a program *satisfies* its specifications if all of its specification sensors give true *whenever* the program runs. *Verifying* a program means showing that its implementation always satisfies its specification.

6.1 Specification Expressions

Specification expressions are boolean expressions about the environment of a program. These expressions are the logical conditions that are expected to be true at specific times as the program runs. The boolean expressions may be annotated with verification directives that indicate how they are to be verified.

```
<specification expression> ::= <simple specification expression>
                             | ( <simple specification expression> )

<simple specification expression> ::=
    <non-validated specification expression> [ <validation directive> ]

<non-validated specification expression> ::=
    [ <proof directive> ] <boolean expression>

<proof directive> ::= PROVE | ASSUME

<validation directive> ::= OTHERWISE <actual condition>
```

The proof directive states whether the specification is to be proved or assumed. `PROVE` is the default if no directive is given. If a validation directive is given, the expression is verified by validating it at run time. In this case, the value of the expression is computed when the program runs, and the actual condition is signalled if the expression is false.

The following subsections describe several features of expressions that are intended primarily for specifications. These features, however, may be used in any expression.

6.1.1 Entry Values

An *entry value* gives the value of an external variable data object at the time the program was called. This provides a way of giving specifications that relate the current values of data objects to values of the external data objects when the program started to run.

```
<entry value> ::= <external variable object>'

<external variable object> ::= <data object name>
```

The object must be a data object that is an external variable object. The object name and the ' mark must be contiguous.

```
Examples:  X'       A'(J)       R'.F(I')
```

6.1.2 Quantified Expressions

Quantified expressions apply the universal and existential quantifiers of ordinary logic to a boolean expression.

```
<quantified expression> ::= <universal quantification>
                          | <existential quantification>

<universal quantification> ::= ALL <bound expression>

<existential quantification> ::= SOME <bound expression>

<bound expression> ::=
    <quantified names> : <type specification>, <boolean expression>

<quantified names> ::= <identifier> { , <identifier> }

<boolean expression> ::= <expression>
```

The appearance of an identifier as a quantified name defines it as a local name (Section 5.3) of the Gypsy unit that contains the quantified expression. A quantified name may be used only within the boolean expression of the quantified expression (as a local constant of type boolean); within the boolean expression, a quantified name may not be used as the quantified name of another quantified expression or as the identifier of an each clause (Section 4.2.4). (If the type specification in a bound expression has an operation restriction on a buffer, the restriction is ignored.)

A *universal quantification* is true if and only if its boolean expression is true for every possible assignment of values of the type specification to the quantified names. An *existential quantification* is true if and only if its boolean expression is true for at least one assignment of values of the type specification to the quantified names.

```
Examples:  ALL I,J : INTEGER, P(I,J,K)
           SOME K:AN_INT(1..10), A(K)=X
```

6.2 External Program Specifications

External operational specifications specify properties about the external environment of a program. An external operational specification may refer only to external objects. It may not refer to internal ones.

```
<external operational specification> ::=
    [ <abstract operational specification> ]
    [ <concrete operational specification> ]
```

```
<abstract operational specification> ::=
    [ <entry specification> ]
    [ <block specification> ]
    [ <exit specification>  ]
```

6.2.1 Entry

An *entry specification* is to be true at the time its program starts running.

```
<entry specification> ::= ENTRY <non-validated specification expression> ;

Examples:  ENTRY N > 0;     ENTRY WELL_FORMED(X,Y);
```

6.2.2 Exit

An *exit specification* is to be true at the time its program stops running (if it does).

```
<exit specification> ::= EXIT <non-validated specification expression> ;
                       | EXIT <conditional exit specification> ;

Examples:  EXIT X = FILTER(Y);
           EXIT ASSUME RESULT = IF N=0 THEN 1 ELSE N*FACTORIAL(N-1) FI;
```

6.3 Internal Program Specifications

Internal specifications are specifications about the total environment of their program. They may refer to both internal and external objects.

6.3.1 Keep

A *keep specification* may be stated after the definition of the internal environment of a program. Once the internal environment is created, the keep must be true throughout the remainder of the running of the program *except* during the running of called programs. However, it must be true immediately before and after the running of each called program. (This applies to both user-defined and pre-defined programs.)

```
<keep specification> ::= KEEP <non-validated specification expression> ;

Example:  KEEP J IN [MIN_J..MAX_J];
```

6.3.2 Assert

An *assert specification* is to be true whenever it is reached as the program runs. (To prove a program with a loop, assert specifications must be placed so that each repetition of each loop encounters at least one assert specification.)

```
<assert specification> ::= ASSERT <specification expression>

Examples:  ASSERT S = SUM(A,1,I-1) & I < N+1
           ASSERT (ASSUME A(K) > 0) OTHERWISE NONPOS_ERROR
```

6.4 Lemma Specifications

A Gypsy *lemma* is a special form for a boolean-valued function that is to be true for all values of its external data objects. The external data objects of a lemma must meet the same requirements as for a function. The RESULT of a lemma is its lemma body, and it must be of type boolean. Thus, the lemma body specifies a relation among the functions it refers to.

```
    <lemma declaration> ::=
        LEMMA <lemma name> [ <external data objects> ] = <lemma body>

    <lemma body> ::= <non-validated specification expression>

    <lemma name> ::= <identifier>

Examples:  LEMMA POSITIVE_BOUND = MAX_N > 0
           LEMMA F_COMMUTES(X,Y:A_MESSAGE) = (ASSUME F(X,Y)=F(Y,X))
```

6.5 Example

The following is an example of specifying and implementing a function F that computes FACTORIAL. The function F is fully specified and implemented. It illustrates an exit specification and an assert specification. The exit specification states that the RESULT of F(N) is equal to FACTORIAL(N). The function FACTORIAL is an example of defining a function solely for specification. The definition is stated in an exit specification with the proof directive ASSUME, and no implementation is given.

```
  SCOPE A =
  BEGIN

    FUNCTION F (N : INTEGER) : INTEGER =
    BEGIN
       EXIT RESULT = FACTORIAL (N);
       VAR I : INTEGER := 1;
       RESULT := 1;
       LOOP
          ASSERT RESULT = FACTORIAL (I - 1) & I > 0;
          RESULT := RESULT * I;
          IF I = N
                THEN LEAVE
             ELSE I := I + 1
          END
       END
    END;

    FUNCTION FACTORIAL (N : INTEGER) : INTEGER =
    BEGIN
       EXIT (ASSUME RESULT = IF N = 0 THEN 1 ELSE N * FACTORIAL (N - 1) FI);
    END;

  END;
```

Chapter 7
SCOPES

All Gypsy units must be grouped textually into one or more scopes. Every unit must appear in some scope. These groupings are a way of assigning names to Gypsy units and of controlling access to those names. Scopes have no other meaning.

A Gypsy *scope* defines a set of local names (Section 7.3) that are used, within the scope, to refer to Gypsy units. It controls its own access to local names in foreign scopes. All local names within a scope must be unique, and all scope names must be unique.

```
<scope declaration> ::=
    SCOPE <scope name> =
    BEGIN
      <unit or name declaration>
      { ; <unit or name declaration> } [ ; ]
    END [;]

<unit or name declaration> ::=
 <unit declaration> | <name declaration>

<scope name> ::= <identifier>
```

7.1 Unit Declaration

A *unit declaration* defines a Gypsy unit -- that is, a type, procedure, function, constant, or lemma. It also defines the name of the unit as a local name of the scope. (The scalar values in a scalar type definition define constant units (Section 3.2.1).)

```
<unit declaration> ::=
      <type declaration>     | <procedure declaration>
    | <function declaration> | <constant declaration>
    | <lemma declaration>
```

7.2 Name Declaration

A *name declaration* is the mechanism for making a unit which was declared in one scope visible in another scope. A name declaration does not define a new Gypsy unit; it defines a new alias for an existing unit.

```
<name declaration> ::= NAME <local aliases> FROM <foreign scope name>

<local aliases> ::= <local renaming> { , <local renaming> }
```

```
<local renaming> ::= [ <local name> = ] <foreign unit name>

<local name> ::= <identifier>

<foreign unit name> ::= <identifier>
```

Each local renaming defines a local alias for the Gypsy unit named by the foreign unit name in the foreign scope. If the local name of a renaming is not given, the local alias is the same as the foreign unit name. The foreign unit name must be the name of a unit declared in the foreign scope. It may not be an alias defined by a name declaration in the foreign scope.

Examples: NAME X FROM PARENT NAME F=G, AN_OBJECT FROM PUBLIC

7.3 Local Names

The local names of a scope are the scope name, the names of the units defined by the unit declarations, and the additional local aliases defined by the name declarations. All local names within a scope must be unique.

7.4 Resolving References

The declaration of a Gypsy unit contains identifiers that refer to data objects, conditions and Gypsy units. There are two sets of local names associated with each unit. One set contains the local names that are internal to unit (Section 5.3). These names refer to data objects and conditions (and they also may be used in each clauses and quantified expressions). The other set contains the local names of the scope in which the unit is declared. These names refer to Gypsy units.

Suppose that an identifier *I* appears in the declaration of a unit *U* which is contained in scope *S*. What *I* refers to is determined by applying the following rules in order.

1. If *I* is an internal local name of unit *U*, then *I* refers to object *I* within *U*.
2. If *I* is a local name of scope *S*, then *I* refers to unit *I* in *S*.
3. If *I* is neither an internal local name of unit *U* nor a local name of scope *S*, the reference is undefined and is not allowed.

Chapter 8
CONDITIONS

Conditions are labels for condition handling statements. *Signalling* a condition causes a forward jump to a condition handler that is labelled with the condition that was signalled.

8.1 Declaring Conditions

Every program has a set of local conditions. The names of these conditions are local names of the program (Section 5.3). (Therefore, every local condition name must be unique within the program.) Only local conditions may be used as labels for condition handlers within the program.

```
<local condition> ::=
    <formal condition name> | <internal condition name>
  | SPACEERROR              | ROUTINEERROR
```

8.1.1 External Conditions

External conditions may be specified in the external environment of a program (procedure or function) declaration (Sections 5.1, 5.4). External conditions are local conditions that refer to conditions in the external environment. SPACEERROR and ROUTINEERROR are declared automatically as formal condition parameters in every program. These conditions have a special meaning, and the user can not signal either of them or use them as actual conditions. They can only be "handled."

```
<external conditions> ::= UNLESS ( COND <formal condition parameters> )

<formal condition parameters> ::=
    <formal condition name> { , <formal condition name> }

<formal condition name> ::= <identifier>
```

```
Example of programs with external conditions:
  PROCEDURE READ_DIGIT(VAR C:CHARACTER) UNLESS (COND NON_NUMERIC) = ...
  FUNCTION GRATE(N:INTEGER):INTEGER UNLESS (COND BAD_N) = ...
```

8.1.2 Internal Conditions

Internal conditions are local conditions that may appear *only* within the internal statements of a procedure body (Section 5.6).

```
<internal condition objects> ::=
    COND <internal condition name> { , <internal condition name> }
```

```
    <internal condition name> ::= <identifier>
Examples:  COND INDEX_ERR, BAD_CHAR
```

8.2 Handling Conditions

Every condition that can be signalled within a program is a local condition (Section 8.5.3). When a condition is signalled, control jumps forward to the *nearest* internal condition handler that is labelled with the condition that was signalled. If there is no such internal handler, then the condition is the name of some formal condition parameter (possibly SPACEERROR or ROUTINEERROR). If so, the (called) program that signalled the condition is terminated, and the corresponding actual condition is signalled in the environment of the calling program.

8.3 Begin Composition

Condition handlers can be associated with any of the procedure composition rules, and a *begin composition* may be used to associate condition handlers with an arbitrary sequence of internal statements.

```
    <begin composition> ::= BEGIN
                            [ <internal statements> ]
                            [ <condition handlers> ]
                            END
```

8.4 Condition Handlers

Condition handlers are internal statements that are performed *only* when conditions are signalled. Conditions handlers can be placed at the end of any procedure composition rule (IF, CASE, LOOP, BEGIN, AWAIT, COBEGIN). A composition rule may have handlers for several conditions. Each handler has one or more labels that identify the handler, and it has a body that consists of internal statements. Each handler label for a single composition rule must be unique (although the same label may be used on different composition rules). If a condition is signalled in the handler body, control jumps to the next condition handler with the matching handler label. It is never handled within the handler which signalled it.

```
    <condition handlers> ::= WHEN { <handler> }

    <handler> ::= IS <handler labels> : <handler body>

    <handler labels> ::= <handler name> { , <handler name> }

    <handler name> ::= <local condition>

    <handler body> ::= [ <internal statements> ]
Example of a begin composition with condition handlers:
  BEGIN
    READ_INPUT(B,IN_FILE) UNLESS (BAD_INPUT);
    COMPUTE_ANSWER(B,OUT_FILE) UNLESS (UNDEFINED);
  WHEN
  IS BAD_INPUT:  PRINT_MESSAGE("INPUT",OUT_FILE);
  IS UNDEFINED:  PRINT_MESSAGE("UNDEFINED",OUT_FILE);
  END
```

8.5 Signalling Conditions

A condition may be signalled in one of several ways. It may be signalled by a signal statement, by a validation directive, or by a procedure or function call.

8.5.1 Forward Conditions

Every condition that is signalled must be a forward condition. A *forward condition* is a formal condition parameter or the name of an internal handler that appears between the point where the condition is signalled and the end of the procedure body. (Neither SPACEERROR nor ROUTINEERROR is a forward condition.)

```
<forward condition> ::= <formal condition name>
                      | <internal condition name>
```

8.5.2 Signal Statement

A signal statement simply signals its forward condition.

```
<signal statement> ::= SIGNAL <forward condition>
```

```
Example:  SIGNAL BAD_INPUT
```

8.5.3 Procedure and Function Calls

Performing a program call may cause any one of its condition parameters to be signalled. A forward condition may be given as an actual parameter for each possible condition parameter that may be signalled. Like data parameters, condition parameters are passed by reference. If a program has formal condition parameters defined, then an actual condition must be supplied for each formal.

```
<actual condition parameters> ::=
    UNLESS ( <actual condition group> )

<actual condition group> ::= [ <group name> ] <actual condition list>

<group name> ::= COND

<actual condition list> ::= <actual condition> { , <actual condition> }

<actual condition> ::= <forward condition>
```

Actual condition parameters are given in five groups: VALUE, ALIAS, COND, SPACE and ELSE. Only the COND group may be supplied explicitly, and the rest are supplied automatically.

The VALUE group contains the conditions that are signalled as a result of the type consistency checks of the actual data parameters (Section 5.9.2.) ROUTINEERROR is signalled if any of the data parameters has a type consistency error.

The ALIAS group contains only a single condition. ROUTINEERROR is signalled if the data parameters contain potentially harmful aliasing (Section 5.9.3.)

The COND group contains the actual conditions that correspond to the explicitly declared COND formal parameters of the program (Section 8.1.1). The COND group must be supplied if there are actual condition names, and there must be exactly one actual for each formal in that group.

The SPACE group contains only a single condition. It is the actual that corresponds to the implicit SPACEERROR formal parameter. SPACEERROR may be signalled by any pre-defined program to indicate that there is insufficient space to continue running the program.

The ELSE group has only a single condition. It is the actual that corresponds to the implicit ROUTINEERROR formal parameter, and it defaults to ROUTINEERROR. ROUTINEERROR is supplied automatically as the actual parameter for every condition in every other group. Thus, ROUTINEERROR denotes some condition (other than SPACEERROR) for which no handler has been specified. (By virtue of these defaults to ROUTINEERROR and SPACEERROR, every condition that is signalled within a program is a local condition of that program.)

```
Example program calls with actual condition parameters:
  P(A,B) UNLESS (COND C4,C5)
  Y := 2 * (SQUARE_ROOT(X) UNLESS (UNDEFINED))
```

8.5.4 Standard Procedures and Functions

Many of the standard functions and procedures are not total, that is there are possible parameter values for which no result can be computed. For example, SCALE(x,BOOLEAN) can only return a value if x is either zero or one. In such a case, ROUTINEERROR is signalled. If a standard operation fails because of system resource failure, SPACEERROR is signalled. (See Appendix C.)

8.6 Conditional Exit Specifications

A *conditional exit specification* allows a separate exit specification to be given for each condition that may be signalled by a program to its external environment.

```
<conditional exit specification> ::=  CASE ( <case exit body> )

<case exit body> ::= <case exit> {; <case exit>}

<case exit> ::= IS <case exit labels> :
                <non-validated specification expression>

<case exit labels> ::= <exit label> { , <exit label> }

<exit label> ::= <formal condition name>
                 | NORMAL | ROUTINEERROR
```

Each case exit label must be unique, and each one identifies one of the ways in which the program can return to its external environment. The non-validated expression that follows the label is the exit specification for that case. NORMAL identifies the case in which no conditions are signalled. (The form EXIT x, described in Section 6.2.2, is an abbreviation for EXIT CASE (IS NORMAL: x).)

```
Example:
  EXIT CASE
       (IS NORMAL: RESULT = BALANCE(N) & N GE 0;
        IS UNDEFINED: N < 0);
```

Chapter 9
DYNAMIC TYPES AND OBJECTS

The dynamic type compositions define values and objects that have a variable number of components. These values and objects can be used in either the specification or the implementation of a program. In an implementation, entire data objects can be created in Gypsy only as internal objects to a program, and once created, an object exists only throughout the running of the program that created it. However, if it is a dynamic object, its *components* can be created, assigned values and deleted dynamically as the program runs. This provides a dynamic object management facility without using explicit pointers.

9.1 Dynamic Type Compositions

The *dynamic type compositions* are sets, sequences and mappings. Each of these compositions have a variable number of components, and each component is of the same type. Each kind of composition is defined by a component type and a size limit restriction. The component type specifies the type of each component, and the *size limit restriction* indicates the maximum number of components. If no size limit restriction is given, there is no limit on the number of components. Each component of a composition is called an *element*.

```
<dynamic type composition> ::=
    <set type> | <sequence type> | <mapping type>

<size limit restriction> ::=
    <non-negative integer pre-computable expression>

  <non-negative integer pre-computable expression> ::=
  <pre-computable expression>
```

9.1.1 Sets

A *set* has a variable number of *unique* elements each of the same type. The default initial value of a set is the empty set.

```
<set type> ::= SET [ ( <size limit restriction> ) ] OF <component type>
```

Each element of a set has a value *v* of the component type, and the value of a set is *set*:{*v1*, ..., *vn*}. There is no way of selecting a particular element of a set. The component type may be any Gypsy type, with the restriction that if the component type is an abstract type, the equality extension function for the type must be defined as concrete equality.

```
Examples of declarations of set types:
  TYPE KEYS = SET (100) OF LARGE_INT
  TYPE PRIMES = SET OF INTEGER
```

9.1.2 Sequences

A *sequence* has a variable number of elements, each of the same type, that are kept in order. The default initial value of a sequence is the empty sequence.

```
<sequence type> ::=
    SEQUENCE [ ( <size limit restriction> ) ] OF <component type>
```

Each element of a sequence has an integer selector *i*, which is the position of that element in the sequence, and a value *v* of the component type. The standard function `SIZE` gives the number of elements in a sequence, and the positions of the elements in a sequence *s* are numbered 1,...,`SIZE`(*s*). It is important to remember that if elements are added to or removed from a sequence, *the position numbers of all succeeding elements will change*. The value of a sequence is *sequence*:{(1,*v1*), ..., (*n*,*vn*)}.

```
Examples of declarations of sequence types:
  TYPE TEAM = SEQUENCE (TEAM_SIZE) OF PLAYER
  TYPE HISTORY = SEQUENCE OF MESSAGE
```

9.1.3 Mappings

A *mapping* has a variable number of elements, each of the same type, that are selected by elements of a selector type rather than by position. The default initial value of a mapping is the empty mapping.

```
<mapping type> ::= MAPPING [ ( <size limit restriction> ) ]
                   FROM <selector type> TO <component type>

<selector type> ::= <equality type>

<equality type> ::= <type specification>
```

An equality type is any type that has equality defined on its value set (Appendix C), except that if the equality type is abstract, its equality extension function must defined as concrete equality. This restriction on equality extension also applies to the component type if it is an abstract type. Each component of a mapping has a *unique* selector *s*, which must be of the selector type, and a value *v* which must be of the component type. The value of a mapping is *mapping*:{(*s1*,*v1*), ..., (*sn*,*vn*)}.

9.2 Expressions

9.2.1 Set and Sequence Values

A *set* or *sequence value* defines the values of a set or a sequence.

```
<set or sequence value> ::= ( <set or seq mark> <element list> )  |
                            ( <range limits> )

<set or seq mark> ::= SET : | SEQ :

<element list> ::= <value list> | <range limits>

<value list> ::= <expression> { , <expression> }
```

If `SET:` is present, the element list defines the elements of a set; otherwise, it defines the elements of a sequence in order from left to right. Range limits define an element list that has one element for each value in the range. (The range may be empty.) If the value list of a set has non-unique values, the set contains only the unique values.

```
Examples: (SET: 'A', 'B', 'C')
          (I..J) {The sequence I, I+1, ..., J}
          (SEQ: F(X), F(Y), F(Z))
```

9.2.2 Component Selectors

There is no way to select an element of a set. The elements of sequences and mappings may be selected with an index in the same way as arrays. A subsequence selector can be applied to a sequence to give the elements within a specific range. A subsequence selector can be used only in a value expression, not in a name expression.

```
    <subsequence selector> ::= <range>

Example subsequence selection:  S(1..10)
```

9.2.3 Operators

Several standard operators are provided for the dynamic types. Their operation is described in Appendix C.

```
    <set operator> ::= EQ | = | NE | IN | ADJOIN | OMIT | SUB
                     | UNION | INTERSECT | DIFFERENCE

    <sequence operator> ::= EQ | = | NE | IN | :> | <: | SUB | @ | APPEND

    <mapping operator> ::= EQ | = | NE | UNION | INTERSECT | DIFFERENCE

Examples:  I IN (M..N)       R @ S       X SUB Y       A UNION B
```

9.2.4 Value Alterations

Value alterations of existing components of sequence values may be made with the ordinary component assignment part of a value alteration (Section 4.2.3). (A value alteration can not be performed on a set because the elements of a set do not have selectors.) A `WITH` clause allows value alterations that create and delete components of values whose type is a dynamic type composition.

```
    <component creation> ::=
        <component creator> <creation component selectors> := <expression>

    <component creator> ::=  BEFORE | BEHIND | INTO

    <component deletion> ::= SEQOMIT <alteration selector list>
                           | MAPOMIT <alteration selector list>

    <creation component selectors> ::= <alteration selector list>
```

A component creation creates a new component of a sequence or mapping. The creation component selectors must designate some component of dynamic value. `BEFORE` and `BEHIND` are used only with selectors that designate an existing component of a sequence. A new component, with the value of the expression, is created either immediately before or behind the component selected.

`INTO` is used to modify mapping values. With an `INTO` creator, the component selectors may designate either an existing or a non-existing component of the mapping. If the component does not exist, it is created; and in either case, the value of the expression is assigned to the (possibly newly created) component.

`SEQOMIT` and `MAPOMIT` designate a sequence or mapping with the selected component deleted from the dynamic value. Only an existing component can be deleted. An attempt to delete a non-existing component

signals a condition.

```
Examples of expressions with value alterations:
  S WITH ( [I] := F(X) )
  S WITH ( BEFORE (I) := F(X) )
  M WITH ( INTO (P+Q) := A & B )
  S WITH ( SEQOMIT (3) )
  M WITH ( MAPOMIT (P+Q) )
```

9.2.5 String Values

A `STRING` is a constant object of type `SEQUENCE OF CHARACTER`.

```
<string value> ::= " { <non-quote character> | <quote symbol> } "

<non-quote character> ::= any character except a "

<quote symbol> ::= ""
```

A string value is the sequence of all ASCII characters that appear *between* (but not including) the opening and closing double quote marks. Within string values, upper and lower case letters are *not* interchangeable, and pairs of parentheses and square brackets are not interchangeable. The quote symbol stands for one double quote character (").

```
Examples:  """" {The string consisting of a single " character}
           "Date of Birth: "
           "         " {The string of 10 blanks}
           """Computo, ergo sum."" {The string "Computo, ergo sum."}
```

9.3 Statements

The following statements dynamically create and delete components of dynamic objects.

9.3.1 New Statement

The *new statement* calls a standard procedure that creates a new component of a dynamic variable object.

```
<new statement> ::= NEW <expression> <new dynamic variable component>

<new dynamic variable component> ::=
    INTO SET <set name expression>
  | INTO <mapping element name expression>
  | BEFORE <sequence position designator>
  | BEHIND <sequence position designator>

<set name expression> ::= <name expression>

<mapping element name expression> ::= <name expression>

<sequence position designator> ::= SEQ <sequence name expression>
                                 | <sequence element name expression>

<sequence name expression> ::= <name expression>

<sequence element name expression> ::= <name expression>
```

The expression defines a new value that is created and assigned to a new component of a dynamic variable. The value of the expression must be of the type of a component of the dynamic variable; otherwise a condition signalled.

A set name expression must name a variable of type set. The new value is put into the set if it is not already there. A mapping element name expression may name either an existing or a non-existing component of a variable of type mapping. If the component already exists, it is assigned the new value just as with an ordinary assignment statement. If the component does not exist, it is created and assigned the new value.

`BEFORE` and `BEHIND` create new components at the designated position in a sequence. A sequence name expression is an expression that names a variable of type sequence. `BEFORE` puts the new component at the beginning of the sequence, and `BEHIND` puts it at the end.

A sequence element name expression is an expression that names some *existing element* of a variable of type sequence. In this case, `BEFORE` puts the new element into the sequence immediately preceding the designated element, and `BEHIND` puts it into the sequence immediately succeeding the designated element. (If the element that is designated does not exist, the component selection signals a condition.)

`SEQ` denotes the whole sequence that is identified by the sequence name expression. For a non-empty sequence *s*, `NEW x BEFORE SEQ s` means the same as `NEW x BEFORE s(1)`. However, if *s* is empty, the `BEFORE SEQ` form creates a single element sequence, whereas the `BEFORE s(1)` form signals a condition. Similarly, for a non-empty *s*, `NEW x BEHIND SEQ s` means the same as `NEW x BEHIND s(SIZE(s))`; and for an empty sequence, the `BEHIND SEQ` form creates a single element sequence, whereas the other form signals a condition.

```
Examples:  NEW F(Z) INTO M(X)              NEW F(Z) INTO SET S
           NEW F(Z) BEFORE S(I)            NEW F(Z) BEFORE SEQ S
           NEW F(Z) BEHIND S(I)            NEW F(Z) BEHIND SEQ S
```

9.3.2 Remove Statement

The *remove statement* calls a standard procedure that deletes one component of a dynamic variable object. A component may be removed from a set, a sequence or a mapping. The component that is removed from the object must exist; otherwise, the component selection signals a condition. (Note that if a sequence element is removed, then the succeeding elements are in a new position.)

```
    <remove statement> ::= REMOVE <removable component>

    <removable component> ::= <set omission>
                            | <dynamic variable component name expression>

  <set omission> ::= ELEMENT <expression> FROM SET <set name expression>

    <dynamic variable component name expression> ::= <name expression>

Examples:  REMOVE ELEMENT X FROM SET S
           REMOVE M(X)                 REMOVE S(I)
```

9.3.3 Move Statement

A *move statement* moves one component of a dynamic variable into a *different* variable. The component may be moved into a new component of a dynamic variable, or it may be assigned to a non-dynamic variable.

```
<move statement> ::= MOVE <removable component> <component destination>

<component destination> ::= <new dynamic variable component>
                          | TO <sequence element name expression>
```

A move statement with a removable component *r* and a new dynamic variable component *d* has the same effect as `NEW` *r d*`;` `REMOVE` *r*. A move statement with a sequence element name expression *s(i)* has the same effect as *s(i)* `:=` *r*`;` `REMOVE` *r*. The move statement can be used only to move a component of a dynamic variable to another *different* dynamic variable. It can not be used to move components within the *same* dynamic variable -- for example, MOVE S(2) BEFORE S(1) is not allowed. (This restriction eliminates some rather peculiar effects that can be caused by aliasing in the call of the standard procedure.)

```
Examples: MOVE ELEMENT X FROM SET S BEHIND T(I)  MOVE T(I) INTO SET S
          MOVE M(X) BEHIND SEQ T                 MOVE T(I) TO S[I]
          MOVE LIVE(KILL) BEFORE SEQ DEAD        MOVE Y INTO M[X]
```

Chapter 10
CONCURRENCY

Gypsy allows several procedures to be concurrently. Each of the procedures that is running concurrently is called a *process*. Message buffers are the only variable objects that can be shared among concurrent processes.

10.1 Buffers

A *buffer type composition* is a special dynamic type whose objects can be shared among concurrent processes. The default initial value is the empty buffer.

```
<buffer type composition> ::=
    BUFFER [ <size limit restriction> ] OF <non-buffer component type>

<non-buffer component type> ::= <type specification>
```

The type specification of the components of a buffer may specify any type that does not contain a buffer. A buffer is a queue of elements of its component type. A buffer object may be passed as a parameter, but the only operations that can modify a buffer queue are the standard send, receive and give procedures.

```
Examples of type declarations of buffer compositions:
  TYPE CHAR_BUF = BUFFER (N) OF CHARACTER
  TYPE DATA = BUFFER OF BIT_BLOCK
```

10.2 Operation Restrictions

The type specification of a buffer may have operation restrictions. An *operation restriction* specifies to what extent a buffer can appear in the standard send, receive and give procedures.

```
<restricted buffer type composition> ::=
    <buffer type name> [ <operation restriction> ]

<buffer type name> ::= <type name>

<operation restriction> ::= "<" <input or output> ">"

<input or output> ::= INPUT | OUTPUT
```

An `INPUT` (only) buffer may appear in a receive statement, but it may not appear in a send or give. An `OUTPUT` (only) buffer may appear in a send or give statement, but it may not appear in a receive statement.

```
Examples:  TYPE IN_BUF = CHAR_BUF <INPUT>
```

10.3 Buffer Parameters

Buffers can be passed as parameters to programs (and created as internal objects), but they can be modified only by the standard send, receive and give statements. The size limit restriction of the actual buffer parameter must be equal to the size limit of its formal parameter. The *type* of the components of the actual must be the *same* as the type of the components of the formal parameter. (Buffer parameters must observe this more severe type restriction because, at the time the call with the buffer parameter is made, it is not possible to know all of the values that might appear as components of the buffer. Therefore, they can not be checked for type consistency at the time of the call.) The formal buffer parameter must be equally or more restricted by operation restrictions than the actual. For example, an unrestricted buffer can be an actual parameter for an input only formal parameter, but not vice versa.

10.4 Statements

The following statements call standard procedures on buffers. These procedures are the only way of modifying a buffer variable. While running, each of them has exclusive access to its buffer parameter, even though the buffer may be shared among other processes.

10.4.1 Receive Statement

A *receive statement* removes the oldest element from the buffer queue and assigns its value to some data object.

```
<receive statement> ::= RECEIVE <name expression> FROM <buffer variable>

<buffer variable> ::= <name expression>
```

The buffer variable is the name of some buffer object. If the buffer queue is empty, the receive call is is blocked (suspended) until some other process sends (or gives) a new element to the buffer.

```
Example:  RECEIVE C(I) FROM IN_BUF
```

10.4.2 Send Statement

A *send statement* puts a new (youngest) element onto the buffer queue.

```
<send statement> ::= SEND <expression> TO <buffer variable>
```

If the buffer queue is full (has a number of elements equal to its size limit restriction), the send call is blocked until some other process receives an element from the buffer.

```
Example:  SEND R.F TO B(OUT)
```

10.4.3 Give Statement

A *give statement* removes a component from a dynamic object and sends it to a buffer.

```
<give statement> ::= GIVE <removable component> TO <buffer variable>
```

A give statement with a removable component *r* and a buffer variable *b* has the same effect as `SEND` *r* `TO` *b*`;` `REMOVE` *r*. The give is blocked until both the send and remove are complete.

```
Example:  GIVE S(TARGET) TO B
```

10.5 Concurrent Composition

The *concurrent compositions* provide mechanisms for performing certain actions concurrently.

10.5.1 Await Composition

The *await composition* provides a way of waiting concurrently on several events to occur.

```
<await composition> ::= AWAIT
                          <await arm> { <await arm> }
                        [ <condition handlers> ]
                        END

<await arm> ::=
    [ <each clause> ] ON <event statement> THEN [ <internal statements> ]

<event statement> ::=
    <send statement> | <receive statement> | <give statement>
```

Each await arm has an event statement and some internal statements. The await composition blocks until at least one of its event statements can be unblocked. Then one of the ones that can be unblocked is selected, it is performed, its associated internal statement are performed, and the await composition is complete.

An await arm that contains an each clause (Section 4.2.4) designates a separate await arm for each value of the identifier in the index type of the each clause. The identifier may be used within the await arm in the same way as an internal constant data object (of the procedure in which the await appears).

```
Example:  AWAIT
            ON RECEIVE C FROM X THEN P(C,Z);
            ON RECEIVE D FROM Y THEN P(D,Z);
          END

          AWAIT
            EACH I:SMALL_INT, ON RECEIVE X FROM B(I) THEN Q(X,B,I);
          END
```

10.5.2 Cobegin Composition

A *cobegin composition* calls several procedures which then run concurrently. It is a generalization of the simple procedure call (Section 5.9). A cobegin of a single procedure has the same effect as a sequential procedure call.

```
<cobegin composition> ::=
    COBEGIN
      <cobegin arm> { ; <cobegin arm> } [;]
    [ <condition handlers> ]
    END

<cobegin arm> ::= [ <each clause> ] <procedure statement>
```

The procedure statements designate the procedures that run concurrently. Each individual procedure statement must satisfy all of the normal parameter passages rules (Sections 5.9, 10.3). In addition, there must be no potentially harmful aliasing (Section 5.9.3) among *any* of the parameters of *any* of the processes. The only objects that are allowed to violate this extended non-aliasing rule are buffers or structured objects consisting solely of buffers.

A cobegin arm containing an each clause (Section 4.2.4) designates a separate process for each value of the identifier in the index type of the each clause. The identifier may be used within the cobegin arm in the same way as an internal constant data object (of the procedure in which the cobegin appears).

The cobegin composition stops only when all of its processes stop. They may stop by running to completion or by signalling a condition. Thus, there is the possibility that a cobegin may stop with several processes having signalled conditions. This is known as a *multiple condition*. A multiple condition can only be handled by a handler for ROUTINEERROR.

```
Example:  COBEGIN                                    COBEGIN
            PRODUCE(A,B);                              TRANSFER(A,B);
            CONSUME(B,C);                              TRANSFER(B,C);
          END                                        END

          COBEGIN
            EACH H:HOST_ID, MULTIPLEX(X(H),Y)
          END
```

10.6 Specifications

Gypsy has a number of special facilities for stating specifications of programs that use buffers.

10.6.1 Type Activationid

The same procedure may appear in more than one arm of a cobegin composition. Thus, the same procedure may be run concurrently by a cobegin as several different processes. To state specifications for concurrent processes, it is necessary to identify each process uniquely. To do this, every Gypsy program has one implicit formal parameter called MYID of type ACTIVATIONID. Some unique value of type ACTIVATIONID is supplied automatically for MYID for every call of every program. Objects of type ACTIVATIONID can be passed as parameters, but the only operations that can be performed on them are "=", "ne", and the standard buffer history functions.

10.6.2 Buffer Histories

For specification of programs that use buffers, Gypsy provides a number of standard *buffer history* functions that give the sequences of values that are sent to a buffer and received from a buffer. These functions provide a way of specifying constraints on the sequences of values that flow through a buffer. These standard functions are described in Appendix C.

```
Examples of standard history function:  INFROM(B,MYID)          OUTTO(B,MYID)
```

10.6.3 Block Specifications

Ordinary entry and exit specifications can be used for procedures that have buffer parameters. However, concurrent processes often are intended never to stop running, and therefore, an exit specification is meaningless. Block specifications provide a way of stating specifications for non-terminating programs that use buffers.

A *block specification* is a specification about the *external* environment of the program that is to be true whenever it is blocked on some buffer operation. A block specification may refer only to external objects. The potential points of blockage are any procedure call with a buffer parameter (including the standard send,

receive, and give procedures) and the await and cobegin compositions. An await composition is *blocked* if and only if all of its event statements are blocked. A cobegin composition is *blocked* if and only if some of it processes have not yet stopped *and* all of those are blocked.

```
    <block specification> ::= BLOCK <non-validated specification expression> ;
```

```
Example:  BLOCK OUTTO(Y,MYID) SUB INFROM(X,MYID);
```

Chapter 11
ABSTRACT TYPES

An *abstract type* is one whose type definition is visible only to certain privileged units. The definitions of ordinary types are visible to all units.

11.1 Type Declaration

An *abstract type declaration* names *all* of the Gypsy units that are privileged to use its type definition. No other units are allowed this privilege. The type definition of an abstract type can be used only by the units named in its privileged units. A scope name in the privileged units list is an abbreviation for every unit that has a local name in that scope.

```
    <abstract type declaration> ::=
        TYPE <type name> [ <default abstract initial value specification> ]
                         <privileged units> = <abstract type body>

    <privileged units> ::= "<" { <unit or scope name> } ">"

    <unit or scope name> ::= <type name> | <procedure name>
                             | <function name> | <constant name>
                             | <lemma name> | <scope name>
```

```
Example:  TYPE WELL_FORMED_STATE <READ, WRITE> =
          BEGIN
            S: SYSTEM_STATE;
            HOLD WELL_FORMED(S);
          END;
```

11.2 Type Body

The *abstract type body* contains the type definition of the abstract type, and it also may have a hold specification.

```
    <abstract type body> ::= <type definition>
                         | BEGIN <identifier> : <type definition> ;
                                 <hold specification>
                           END
```

The identifier is a local name of the type, and it must satisfy the restrictions for local names (See 5.3). The identifier is a name for an arbitrary value of that type, and it may be referred to in the hold specification. The type definition defines the *concrete values* of an abstract type.

11.3 Equality Extension

An abstract type also defines a set of abstract values. An *equality extension* for the abstract type defines the equality of abstract values. An equality extension function must be defined for every abstract type. Its name must appear on the access list for the abstract type.

```
<equality extension> ::= EXTENDS "="
```

The equality extension is provided as part of a function declaration (Section 5.4). The function must have exactly two formal constant parameters of the abstract type, and its result type may be boolean. (Only one equality function may be defined for an abstract type.) When the = (or EQ) operator is applied to values of the abstract type in units that are not privileged to use its type definition, the equality extension function for that type is applied. Within a privileged unit, the = operator is treated in the usual way.

Defining equality of the abstract values partitions the concrete values into equivalence classes. (Every concrete value is in exactly one equivalence class, but one equivalence class may contain several concrete values.) Each equivalence class represents a different abstract value. The equality extension defines when two concrete values are in the same equivalence class, or in other words, when they represent the same abstract value.

```
Example:  FUNCTION WELL_EQ EXTENDS "="(P,Q:WELL_FORMED_STATE):BOOLEAN = ....
```

11.4 Specifications

11.4.1 Default Initial Values

An abstract *default initial value specification* of an abstract type may be stated.

```
<default abstract initial value specification> ::=
    INITIALLY [ <proof directive> ] <expression>
```

The expression must be of the abstract type.

```
Example:  TYPE WELL_FORMED_STATE INITIALLY INIT_STATE <READ, WRITE> = ....
```

11.4.2 Hold

A *hold specification* of an abstract type is a relation that is to be true of every abstract object upon completion of every program that is privileged to use the type definition of the abstract type. It also must be true when an abstract object is created. In effect, a hold specification selects a subset of the concrete values that can be used to represent abstract values.

```
<hold specification> ::= HOLD <specification expression> [;]
```

```
Example:  HOLD WELL_FORMED(S);
```

11.4.3 Centry, Cblock, Cexit

The *concrete specifications* of a program may use the type definition of an abstract type if their program is privileged to use that definition. This is in direct contrast to the abstract specifications of the program. An *abstract specification* may not use the type definition of any abstract type, even if its program does have that privilege. Abstract specifications always must be stated strictly in abstract terms. The concrete specifications have the same meaning as their abstract counterparts.

```
<concrete operational specification> ::=
```

```
    [ <concrete entry specification> ]
    [ <concrete block specification> ]
    [ <concrete exit specification>  ]

<concrete entry specification> ::=
    CENTRY <non-validated specification expression> ;

<concrete block specification> ::=
    CBLOCK <non-validated specification expression> ;

<concrete exit specification> ::=
    CEXIT <non-validated specification expression> ;
    | CEXIT <conditional exit specification> ;
```

11.4.4 Lemmas

A lemma may be a privileged unit of an abstract type. As with the abstract operational specifications, the lemma body may *not* use the type definition of an abstract type even if the lemma has that privilege. The body must be stated in purely abstract terms. The privilege of using the type definition, however, may be exercised in *proving* the lemma.

Appendix A
Reserved Identifiers

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, 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.

Words Reserved for Language Extensions

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

Standard Types

ACTIVATIONID, BOOLEAN, CHARACTER, INTEGER, RATIONAL.

Boolean Values

TRUE, FALSE.

Functions

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.

Conditions

ROUTINEERROR, SPACEERROR.

Appendix B
Base Type Definitions

For type `t`, its base type, `btype(t)`, is defined as shown below. If type `t` has a composition access list, its base type has the same list.

TYPE t<access list>=...	**btype(t)<access list>=...**
v	**btype(v)**
v(range restriction)	**btype(v)**
v<buffer restriction>	**btype(v)**
ARRAY u of v	**ARRAY u of btype(v)**
RECORD(f1:u1;...;fn:un)	**RECORD(f1:btype(u1);...fn:btype(un))**
SET OF v **SET (size restriction) OF v**	**SET OF btype(v)** **SET OF btype(v)**
SEQUENCE OF v **SEQUENCE (size restriction) OF v**	**SEQUENCE OF btype(v)** **SEQUENCE OF btype(v)**
MAPPING FROM u TO v **MAPPING (size restriction) FROM u TO v**	**MAPPING FROM btype(u) TO btype(v)** **MAPPING FROM btype(u) TO btype(v)**
BUFFER OF v **BUFFER (size restriction) OF v**	**BUFFER OF btype(v)** **BUFFER (size restriction) OF btype(v)**

Appendix C
Standard Operators and Functions

This appendix lists all of the standard Gypsy operators and functions. A syntax example for each operation is given, along with the type requirements of the operands and a brief description what the operation does.

No standard functions have formal condition parameters. The only conditions which may be signalled out of a standard function or a predefined Gypsy operation are spaceerror, which indicates a deficiency of resources in the computing environment, and routineerror, which signifies any other situation under which the operation was unable to return normally. Such situations might include arithmetic overflow, cases where the function is not complete over its domain, or any other case where the correct result cannot be returned. Where a function or operation is not complete, the description below notes those specific cases where routineerror will certainly be signalled.

OPERATION	TYPE REQUIREMENTS	EFFECT
x ** y	Btype(x) = INTEGER. Btype(y) = INTEGER. Result INTEGER.	x to the power y. Signals ROUTINEERROR if y<0, if x = 0 and y = 0, or in case of arithmetic overflow.
	Btype(x) = RATIONAL. Btype(y) = INTEGER. Result RATIONAL.	Same as above.
- x	Btype(x) = INTEGER. Result INTEGER.	Negative x. Signals ROUTINEERROR in case of arithmetic overflow.
	Btype(x) = RATIONAL. Result RATIONAL.	Same as above.
x * y	Btype(x) INTEGER. Btype(y) = INTEGER. Result INTEGER.	x times y. Signals ROUTINEERROR in case of arithmetic overflow.
	Btype(x) = RATIONAL. Btype(y) = RATIONAL. Result RATIONAL.	Same as above.
x / y	Btype(x) = INTEGER. Btype(y) = INTEGER. Result RATIONAL.	x divided by y. Signals ROUTINEERROR if y = 0 or in case of arithmetic overflow.
	Btype(x) = RATIONAL. Btype(y) = RATIONAL. Result RATIONAL.	Same as above.
x DIV y	Btype(x) = INTEGER. Btype(y) = INTEGER. Result INTEGER.	Integer quotient of x divided by y. Signals ROUTINEERROR if y = 0 or in case of arithmetic overflow.

x MOD y	**Btype(x) = INTEGER.** **Btype(y) = INTEGER.** **Result INTEGER.**	**Integer remainder of x divided by y.** **(x DIV y) * y + x MOD y = x.** **Signals ROUTINEERROR if y = 0 or in case of arithmetic overflow.**
x + y	**Btype(x) = INTEGER.** **Btype(y) = INTEGER.** **Result INTEGER.**	**x plus y.** **Signals ROUTINEERROR in case of arithmetic overflow.**
	Btype(x) = RATIONAL. **Btype(y) = RATIONAL.** **Result RATIONAL.**	**Same as above.**
x - y	**Btype(x) = INTEGER.** **Btype(y) = INTEGER.** **Result INTEGER.**	**x subtract y.** **Signals ROUTINEERROR in case of arithmetic overflow.**
	Btype(x) = RATIONAL. **Btype(y) = RATIONAL.** **Result RATIONAL.**	**Same as above.**
x <: y	**Btype(x) = SEQUENCE OF btype(y).** **Result btype(x).**	**x @ (SEQ: y).**
x :> y	**Btype(y) = SEQUENCE OF btype(x).** **Result btype(y).** **(:> is right associative.)**	**(SEQ: x) @ y.**
x ADJOIN y	**Btype(x) = SET OF btype(y).** **Result btype(x).**	**x UNION (SET: x)**
x OMIT y	**Btype(x) = SET OF btype(y).** **Result btype(x).**	**x with element y removed.** **Signals ROUTINEERROR if y is not in x.**
x @ y **x APPEND y**	**Btype(x) = SEQUENCE OF t.** **Btype(y) = SEQUENCE OF t.** **Result btype(x).**	**The sequence x followed by sequence y.**
x UNION y	**Btype(x) = SET OF t.** **Btype(y) = SET OF t.** **Result btype(x).**	**Contains the elements that are in either x or y.**
	Btype(x) = MAPPING FROM u TO v. **Btype(y) = MAPPING FROM u TO v.** **Result btype(x).**	**Contains the (p,q) components that are in either x or y.** **Signals ROUTINEERROR if there are components (p,q1) and (p,q2) with q1 NE q2.**
x INTERSECT y	**Btype(x) = SET OF t.** **Btype(y) = SET OF t.** **Result btype(x).**	**Contains the elements that are in both x and y.**

```
                Btype(x) = MAPPING          Contains the (p,q) components
                  FROM u TO v.              that are in both x and y.
                Btype(y) = MAPPING          Signals ROUTINEERROR if there are
                  FROM u TO v.              components (p,q1) and (p,q2)
                Result btype(x).            with q1 NE q2.

x DIFFERENCE y
                Btype(x) = SET OF t.        Contains the elements that
                Btype(y) = SET OF t.        are in x by not in y.
                Result btype(x).

                Btype(x) = MAPPING          Contains the (p,q) components
                  FROM u TO v.              that are in x but not in y.
                Btype(y) = MAPPING          Signals ROUTINEERROR if there are
                  FROM u TO v.              components (p,q1) and (p,q2)
                Result btype(x).            with q1 NE q2.

x = y           Btype(x) = btype(y).
x EQ y          Result BOOLEAN.

                Btype(x) = scalar type.     TRUE if x is the same scalar
                                            value as y; otherwise FALSE.

                Btype(x) = INTEGER.         TRUE if x is the same integer
                                            number as y; otherwise FALSE.

                Btype(x) = RATIONAL.        TRUE if x is the same rational
                                            number as y; otherwise FALSE.
                                            (1/2 = 2/4 = 3/6 = etc.)

                Btype(x) = ACTIVATIONID     TRUE if x and y indicate the
                                            same activation of the same
                                            procedure; otherwise FALSE.

                Btype(x) = ARRAY u OF v.    TRUE if x(i) = y(i) for each i
                x and y must have the       in the index range; otherwise
                same index type.            FALSE.

                Btype(x) = RECORD(...).     TRUE if x.f = y.f for each
                x and y must have the       field f; otherwise FALSE.
                same field names.

                Btype(x) = SET OF v.        TRUE if x and y have the same
                                            elements; otherwise FALSE.

                Btype(x) = SEQUENCE OF v.   TRUE if SIZE(x) = SIZE(y)
                                            and x(i) = y(i) for all i
                                            in 1..SIZE(x);otherwise FALSE.

                Btype(x) =                  TRUE if DOMAIN(x) = DOMAIN(y)
                  MAPPING FROM u TO v.      and x(i) = y(i) for all i
                                            in DOMAIN(x); otherwise FALSE.

x NE y          Btype(x) = btype(y).        NOT x = y.
                Result BOOLEAN.

x < y           Btype(x) = btype(y).
```

x LT y	**Result BOOLEAN.**	
	Btype(x) scalar.	**TRUE if ORD(x) < ORD(y); otherwise FALSE.**
	Btype(x)=INTEGER.	**TRUE if x is less than y; otherwise FALSE.**
	Btype(x) RATIONAL.	**Same as above.**
x LE y	**Btype(x) = btype(y).** **Btype(x) simple type.** **Result BOOLEAN.**	**x < y OR x = y.**
x > y **x GT y**	**Btype(x) = btype(y).** **Btype(x) simple type.** **Result BOOLEAN.**	**y < x.**
x GE y	**Btype(x) = btype(y).** **Btype(x) simple type.** **Result BOOLEAN.**	**x > y OR x = y.**
x IN y	**Btype(y) =** **SEQUENCE OF btype(x).** **Result BOOLEAN.**	**TRUE if x is an element of sequence y; otherwise FALSE.**
	Btype(y) = **SET OF btype(x).** **Result BOOLEAN.**	**TRUE if x is an element of set y; otherwise FALSE.**
x SUB y	**Btype(x) = SET OF t.** **Btype(y) = SET of t.** **Result BOOLEAN.**	**TRUE if x is a subset of y; otherwise FALSE.**
	Btype(x) = SEQUENCE OF t. **Btype(y) = SEQUENCE of t.** **Result BOOLEAN.**	**TRUE if y has an ordered (not necessarily contiguous) subsequence x; otherwise FALSE.**
	Btype(x) = MAPPING **FROM u TO v.** **Btype(y) = MAPPING** **FROM u TO v.** **Result BOOLEAN.**	**TRUE if DOMAIN(x) SUB DOMAIN(y) and x(d) = y(d) for every d IN DOMAIN(x); otherwise FALSE.**
NOT x	**Btype(x) = BOOLEAN.** **Result BOOLEAN.**	**TRUE if x = FALSE;** **FALSE if x = TRUE;**
x & y **x AND y**	**Btype(x) = BOOLEAN.** **Btype(y) = BOOLEAN.** **Result BOOLEAN.**	**TRUE if both x = TRUE and y = TRUE; otherwise FALSE.**
x OR y	**Btype(x) = BOOLEAN.** **Btype(y) = BOOLEAN.** **Result BOOLEAN.**	**TRUE if either x = TRUE or y = TRUE; otherwise FALSE.**
x -> y **x IMP y**	**Btype(x) = BOOLEAN.** **Btype(y) = BOOLEAN.**	**FALSE if x = TRUE and y = FALSE;** **otherwise TRUE.**

```
              Result BOOLEAN.


x IFF y       Btype(x) = BOOLEAN.       x = y.
              Btype(y) = BOOLEAN.
              Result BOOLEAN.

ALL x : xtype, p(x)
              xtype = a bounded simple  This is a shorthand notation for
                type.                   the expression:
              btype(p(x)) = BOOLEAN.    p(x1) & p(x2) & ... & p(xn),
              p(x) is an expression.    where (x1, x2, ... xn) is the
                                        value set of type xtype.

SOME x : xtype, p(x)
              xtype = a bounded simple  This is a shorthand notation for
                type.                   the expression:
              btype(p(x)) = boolean.    p(x1) or p(x2) or ... or p(xn),
              p(x) is an expression.    where (x1, x2, ... xn) is the
                                        value set of type xtype.

x(y)          Btype(x) =                Component y of x.
                ARRAY y OF btype(v).    Signals ROUTINEERROR if there
              Btype(y) = non-rational   is no such component.
                simple type.
              Result type v.

              Btype(x) = SEQUENCE OF v. Same as above.
              Result type v.

              Btype(x) =                Same as above.
                MAPPING FROM u TO v.
              Result type v.

x(y..z)
              Btype(x) = SEQUENCE OF v. If y le z, the subsequence of x that
              Btype(y) = INTEGER.       begins with element in position y and
              Btype(z) = INTEGER.       ends with the element in position z.
              Result type x.            If y > z, produces the empty sequence
                                        of type btype(x).
                                        Signals ROUTINEERROR if y LE z and
                                        there is either no y or no z component.
x.y           Btype(x) =                Field y of record x.
                RECORD(... y:v ...).
              Result type v.

(SET: x1, ..., xn)
              Btype(x1)=...=btype(xn).  The set of elements {x1, ..., xn}.
              Result SET OF btype(x1).

(SET: x..y)
              Btype(x) = btype(y) =     The set of elements {x, ..., y}.
                simple non-rational
                type.
              Result SET OF btype(x).

(SEQ: x1, ..., xn)
```

```
(x1, ..., xn)
              Btype(x1)=...=btype(xn).  The sequence of elements
                                        (x1, ..., xn)
              Result SEQUENCE
                       OF btype(x1).

(SEQ: x..y)
(x..y)
              Btype(x) = btype(y) =     The sequence of elements
                simple non-rational     (x, ..., y)
                type.
              Result SEQUENCE
                       OF btype(x).

x WITH ((y) := z)
              Btype(x) =                Same value as x but with
               ARRAY u OF btype(z).     x(y) = z.
              Result btype(x).          Signals ROUTINEERROR if there is
                                        no y component of x.

              Btype(x) =                Same as above.
               SEQUENCE OF btype(z).
              Btype(y)=INTEGER.
              Result btype(x).


x WITH (.y := z)
              Btype(x) = RECORD         Same value as x but with
                (... y:btype(z) ...).   x.y = z.
              Result btype(x).

x WITH (INTO (y) := z)                  Same value as x but with
              Btype(x) = MAPPING        x(y) = z.
                FROM btype(x)           If there is no
                TO btype(y).            component y, one is created with
              Result btype(x).          z as its value.

x WITH (BEFORE (y) := z)
              Btype(x) = SEQUENCE       x(1..y-1) @ (SEQ: z)
                       OF btype(z).                 @ x(Y..SIZE(x))
              Btype(y) = INTEGER.       Signals routtineerror if there is
              Result btype(x).          no y component of x.

x WITH (BEHIND (y) := z)
              Btype(x) = SEQUENCE       x(1..y) @ (SEQ: z)
                       OF btype(z).                 @ x(y+1..SIZE(x))
              Btype(y) = INTEGER.       Signals ROUTINEERROR if there is
              Result btype(x).          no y component of x.

x WITH (SEQOMIT (y))
              Btype(x) = SEQUENCE OF v. x(1..y-1) @ x(y+1..SIZE(x))
              Btype(y) = INTEGER.       Signals ROUTINEERROR if there
              Result type of x.         is no y component.

x WITH (MAPOMIT (y))
              Btype(x) = MAPPING        Same value as x but without
                FROM btype(y) TO v.     component (y,x(y)).
```

```
              Result type of x.          Signals ROUTINEERROR if there is
                                         no y component.

x WITH (<component selector 1> <component selector 2> := y)
              Must meet the type         x WITH (<component selector 1>
              requirements of its         := x<component selector 1>
              result expression.             WITH (<component selector 2> := y))
              Result btype(x).

x WITH (y ; z)
              Must meet the type         (x WITH y) WITH z
              requirements of its
              result expression.
              Result btype(x).

x WITH (EACH y:t, z)
              Type T is a simple bounded
              type with values ranging
              from a to b.
              Must meet the type         x WITH (a replacing y in z;
              requirement of its                ...
              result expression.                b replacing y in z)
              Result btype(x).

IF   x1 THEN y1
ELIF x2 then y2
...
ELIF xn THEN yn
ELSE z FI
              Btype(x1) = ... =          y1 if x1 = TRUE; otherwise
                btype(xn) =  BOOLEAN.    y2 if x2 = TRUE; otherwise
              Btype(y1) = ... =           ...
               btype(yn) = btype(z).     yn if xn = TRUE; otherwise z
              Result btype(z).

ALLFROM(x)    Btype(x) = BUFFER OF t.    MESSAGES (TIMEDALLFROM (x)).
              Result SEQUENCE OF t.

ALLTO(x)      Btype(x) = BUFFER OF t.    MESSAGES (TIMEDALLTO (x)).
              Result SEQUENCE OF t.

CONTENT(x)    Btype(x) = BUFFER OF t.    The sequence of all values
              Result SEQUENCE OF t.      in the queue of x.
                                         ALLTO(x) = ALLFROM(x) @ CONTENT(x).

DOMAIN(x)     Type of x is               The set of all p such that
              MAPPING FROM u TO v.       (p,q) is in x.
              Result type SET OF u.

EMPTY(x)      Btype(x) = BUFFER OF t.    SIZE(CONTENT(x))=0.
              Result BOOLEAN.

FIRST(x)      Btype(x) = SEQUENCE OF v.  x(1)
              Result type v.             Signals ROUTINEERROR if SIZE(x) = 0.

FULL(x)       Btype(x) = BUFFER OF t.    SIZE (CONTENT(x)) = k where is the
              Result BOOLEAN.            size limit restriction on the buffer.
```

```
                                        FALSE if there is no size limit.

INFROM(x,y)     Btype(x) = BUFFER OF t.  MESSAGES (TIMEDINFROM (x,y)).
                Btype(y) = ACTIVATIONID.
                Result SEQUENCE OF t.

INFROMMERGE(x,y,p,q)                     MESSAGES (TIMEDINFROMMERGE (x,y,p,q))
                Btype(x) = BUFFER OF v.
                Btype(y) = ARRAY INTEGER OF ACTIVATIONID.
                Btype(p) = btype(q) = INTEGER.
                Result SEQUENCE OF v.

INITIAL(x)      x is an identifier that  The default initial value of
                  names a type.          type x.
                Result type x.

LAST(x)         Btype(x) =               x [SIZE(x)].
                  SEQUENCE of v.         Signals ROUTINEERROR if SIZE(x) = 0.
                Result type v.

LOWER(x)        x = the type name of any The minimum value of type x.
                simple type except
                unbounded INTEGER or
                RATIONAL.
                Result type x.

MAX(x,y)        Btype(x) = btype(y).     IF x > y THEN x ELSE y FI.
                Btype(x) = a simple type.
                Result btype(x).

MIN(x,y)        Btype(x) = btype(y).     IF x < y THEN x ELSE y FI.
                Btype(x) = a simple type.
                Result btype(x).

MESSAGES(x)     Btype(x) = SEQUENCE OF t (SEQ: x(1).MESSAGE,...,
                where TYPE t = RECORD          x(SIZE(x)).MESSAGE)
                   (MESSAGE: btype(v);
                    TIME: INTEGER).
                type v a non-buffer type.

NONFIRST(x)     Btype(x) = SEQUENCE OF v. x(2..SIZE(x))
                Result type x.           Signals ROUTINEERROR if SIZE(x) = 0.

NONLAST(x)      Btype(x) =               x(1..SIZE(x)-1).
                  SEQUENCE OF v.         Signals ROUTINEERROR if SIZE(x) = 0.
                Result type x.

NULL(x)         x is an identifier       The empty set of type x.
                  that names a set type.
                Result type x.

                x is an identifier       The empty sequence of type x.
                  that names a sequence
                  type.
                Result type x.

                x is an identifier       The empty mapping of type x.
```

```
                that names a mapping
                type.
              Result type x.

ORD(x)        Btype(x) = a scalar type  The number of the position of
              Result INTEGER.           scalar value x in its base type
                                        definition sequence (with
                                        numbering beginning at zero).

OUTTO(x,y)    Btype(x) = BUFFER OF t.   MESSAGES (TIMEDOUTTO(x,y))
              Btype(y) = ACTIVATIONID.
              Result SEQUENCE OF t.

OUTTOMERGE(x,y,p,q)                     MESSAGES (TIMEDOUTTOMERGE(x,y,p,q))
              Btype(x) = BUFFER OF v.
              Btype(y) = ARRAY INTEGER OF ACTIVATIONID.
              Btype(p) = btype(q) = INTEGER.
              Result SEQUENCE OF v.

PRED(x)       Btype(x) = t where t is   The next scalar value less than x.
                a scalar type.          Signals ROUTINEERROR if x = LOWER(t).
              Result btype(x).

RANGE(x)      Btype(x) =                The set of all q such that
                MAPPING FROM u TO v.    (p,q) is in x.
              Result type SET OF v.

SCALE(x,y)    Btype(x) = INTEGER.       Scalar value number x of type btype(y).
              y is an identifier        Signals ROUTINEERROR if x < 0
              that names a scalar type.   or if x > ORD (UPPER(btype(y))).
              Result type btype(y).

SIZE(x)       Btype(x) = SET OF v.      The number of elements in x.
              Result INTEGER.

              Btype(x) = SEQUENCE OF v. Same as above.
              Result INTEGER.

              Btype(x) = MAPPING        Same as above.
                       FROM u TO v.
              Result INTEGER.

SUCC(x)       Btype(x) = t where t is   The next scalar value greater
                a scalar type.          than x.
              Result btype(x).          Signals routineeror if x = UPPER(t).

TIMEDALLFROM(x)
              Btype(x) = BUFFER OF v.   TIMEDINFROM (x,p) where p is
              Result SEQUENCE OF t      the process in which buffer x
              where TYPE t = RECORD     defined as internal object.
                 (MESSAGE: btype(v);
                  TIME: INTEGER).


TIMEDALLTO(x)
              Btype(x) = BUFFER OF v.   TIMEDOUTTO (x,p) where p is
              Result SEQUENCE OF t      the process in which buffer x
```

```
              where TYPE t = RECORD     is defined as internal object.
                 (MESSAGE: btype(v);
                  TIME: INTEGER).

TIMEDINFROM(x,y)                        TIMEDINFROM (x,y)(k).MESSAGE
              Btype(x) = BUFFER OF v.   is k-th value that was received
              Btype(y) = ACTIVATIONID.  from buffer x by process y.
              Result SEQUENCE OF t      TIMEDINFROM (x,y)(k).TIME
              where TYPE t = RECORD     is the time at which process y
                 (MESSAGE: btype(v);    obtained the k-th value from
                  TIME: INTEGER).       buffer x.


TIMEDINFROMMERGE(x,y,p,q)               TIMEDMERGE (TIMEDINFROM (x,y(p)),
              Btype(x) = BUFFER OF v.                ...,
              Btype(y) =                             TIMEDINFROM (x,y(q)))
                ARRAY INTEGER OF        where
                    ACTIVATIONID.       TIMEDMERGE (u,...v,w) =
              Btype(p) = btype(q)         TIMEDMERGE (u,TIMEDMERGE (...v,w))
                       = INTEGER.
              Result SEQUENCE OF t
              where TYPE t = RECORD
                 (MESSAGE: btype(v);
                  TIME: INTEGER).

TIMEDMERGE(x,y)                         Consists of the elements of the
              Btype(x)=btype(y).        sequences x and y ordered on their
              Btype(x) = SEQUENCE OF t  TIME components.
              where TYPE t = RECORD     (TIMEDMERGE (x,y) is defined only if
                 (MESSAGE: v;           TIMEDORDER (x) and TIMEDORDER (y)
                  TIME: INTEGER).       and each TIME component is unique
              Result btype(x).          in both sequences.)

TIMEDORDER(x)                           TRUE if x(k).TIME < x(k+1).TIME
              Btype(x) = SEQUENCE OF t  for k = 1,...,SIZE(x)-1.
              where TYPE t = RECORD     FALSE otherwise.
                (MESSAGE: v;
                 TIME: INTEGER).
              Result BOOLEAN.

TIMEDOUTTO(x,y)                         TIMEDOUTTO (x,y)(k).MESSAGE
              Btype(x) = BUFFER OF v.   is k-th value that was sent (or given)
              Btype(y) = ACTIVATIONID.  to buffer x by process y.
              Result SEQUENCE OF t      TIMEDOUTTO (x,y)(k).TIME
              where                     is the time at which process y
                TYPE t = RECORD         sent (or gave) the k-th value from
                 (MESSAGE: btype(v);     buffer x.
                  TIME: INTEGER).


TIMEDOUTTOMERGE(x,y,p,q)                TIMEDMERGE (TIMEDOUTTO (x,y(p)),
              Btype(x) = BUFFER OF v.                ...,
              Btype(y) =                             TIMEDOUTTO (x,y(q)))
                ARRAY INTEGER OF        where
                    ACTIVATIONID.       TIMEDMERGE (u,...v,w) =
              Btype(p) = btype(q)         TIMEDMERGE (u,TIMEDMERGE (...v,w))
```

```
                = INTEGER.
          Result SEQUENCE OF t
          where TYPE t = RECORD
             (MESSAGE: btype(v);
              TIME: INTEGER).

UPPER(x)  x = the type name of any  The maximum value of type x.
          simple type except
          unbounded INTEGER or
          RATIONAL.
          Result type x.
```

Appendix D
Standard Procedures

PROCEDURE	TYPE REQUIREMENTS	EFFECT
x := y		(Section 5.8.1)
	Btype(x) = btype(y).	Makes the value of x equal to the
	Btype(x) may be any	value of y. As a result of the
	type except	assignment x = y.
	BUFFER OF v.	Signals ROUTINEERROR if y is not in
		the value set of the type of x.
x(y) := z	Btype(x) =	(x := x with ((y) := z))
	ARRAY u OF v.	Signals ROUTINEERROR if y is not in
	Btype(y) = btype(u).	the value set of the index type of x
	Btype(z) = btype(v).	or if z is not in the value set of the
		element type of x.
	Btype(x) =	Same as above.
	SEQUENCE OF v.	Signals ROUTINEERROR if y is not in the
	Btype(y) = INTEGER.	range [1..SIZE(x)] or if z is not in the
		value set of the element type of x.
	Btype(z) = btype(v).	
(x.y := z)	Btype(x) =	x := x WITH (.y := z)
	RECORD(.. y:v ..).	Signals ROUTINEERROR if z is not in
	Btype(z) = v.	the value set of x.y.
NEW z BEFORE SEQ x;		x := z :> x
	Btype(x) =	Signals ROUTINEERROR if z is not in
	SEQUENCE OF v.	the value set of the element type
	Btype(z) = v.	of x or if SIZE(x) = the size
		restriction on the type of x.
		(Section 9.3.1).
NEW z BEHIND SEQ x;		x := x <: z;
	Btype(x) =	Signals ROUTINEERROR if z is not in
	SEQUENCE OF v.	the value set of the element type
	Btype(z) = v.	of x or if SIZE(x) = the size
		restriction on the type of x.
		(Section 9.3.1).
NEW z BEFORE x(y)		x := x WITH (BEFORE (y) := z)
	Btype(x) =	Signals ROUTINEERROR if y is not in
	SEQUENCE of v.	[1..SIZE(X)], if z is not in the
	Btype(y) = INTEGER.	value set of the element type of x,
	Btype(z) = v.	or if SIZE(x) = the size restriction
		on the type of x. (Section 9.3.1).
NEW z BEHIND x(y)		x := x WITH (BEHIND (y) := z)
	Btype(x) =	Signals ROUTINEERROR if y is not in
	SEQUENCE of v.	[1..SIZE(X)], if z is not in the

	Btype(y)=INTEGER. Btype(z)=v.	value set of the element type of x, or if SIZE(x) = the size restriction on the type of x. (Section 9.3.1).
NEW z INTO x NEW z INTO SET x;	Btype(x) = SET OF v. Btype(z) = v.	x := x ADJOIN z Signals ROUTINEERROR if z is not in the value set of the element type of x or if SIZE(x ADJOIN z) > the size restriction on the type of x. (Section 9.3.1).
NEW z INTO x(y)	Btype(x) = MAPPING FROM u TO v. Btype(y) = u. Btype(z) = v.	x := x WITH (INTO (y) := z) Signals ROUTINEERROR if y is not in the value set of the domain type of x, if z is not in the value set of the range type of x, or if SIZE(x WITH (INTO (y) := z)) > the size restriction on the type of x. (Section 9.3.1).
REMOVE x(y)	Btype(x) = SEQUENCE OF v. Btype(y) = INTEGER.	x := x WITH (SEQOMIT (y)); Signals ROUTINEERROR if y is not in [1..SIZE(x)]. (Section 9.3.2).
	Btype (x) = MAPPING FROM u TO v. Btype(y) = u.	x := x WITH (MAPOMIT (y)) Signals ROUTINEERROR if y is not in DOMAIN(x). (Section 9.3.2).
REMOVE ELEMENT z FROM x REMOVE ELEMENT z FROM SET x	Btype(x) = SET OF v. Btype(z) = v.	x := x OMIT z Signals ROUTINEERROR if x is not in x. (Section 9.3.2).
MOVE x y	Same restrictions as equivalent NEW and REMOVE.	Same effect as NEW x y; REMOVE x. (Section 9.3.3).
MOVE x TO y	Same restrictions as equivalent := and REMOVE.	Same effect as y := x; REMOVE x. (Section 9.3.3).
RECEIVE x FROM y	Btype(y) = BUFFER OF v. Btype(x) = v.	Takes oldest value from the queue of y and assigns it to x. Signals ROUTINEERROR if the value is not in the value set of x. (Section 10.4.1).
SEND x TO y	Btype(y) = BUFFER OF v. Btype(x) = v.	Puts a new value on the queue of y. Signals ROUTINEERROR if x is not in the value set of the component type of y. (Section 10.4.2).
GIVE x TO y	Btype(y) = BUFFER OF v. Btype(x) = v.	Same effect as SEND x TO y; REMOVE x. (Section 10.4.3).

Appendix E
Procedure Compositions

In the following, "{ x }" means that "x" may be repeated zero or more times, and "[x]" means that " x " is optional. "Statements" or "statements1", etc. are <internal statements>.

IF bool1 THEN [statements1] ** {ELIF bool2 THEN [statements2]}** ** [ELSE [statements3]]** ** [WHEN {IS condi: statementsi}]** **END**	**An if composition chooses and performs one of several internal statement lists. (Section 5.8.3).**
CASE exp ** {IS labeli: [statementsi]}** ** [ELSE: [statementsj]]** ** [WHEN {is condi: statementsi}]** **END**	**A case composition is another way of choosing and performing one of several internal statement lists. (Section 5.8.4).**
LOOP [statements] ** [WHEN {is condi: statementsi}** **END** **LEAVE**	**A loop composition performs its internal statements repeatedly. It is terminated by performing a leave statement (or by signalling a condition. (Section 5.8.5).**
procname (exp1, .. expn) ** [UNLESS [(COND ccond1, .. ccondn)]**	**The procedure call causes a procedure to run. The actual parameters of the call are objects in the calling environment which become the external objects of the called procedure. Call by reference is used. Actual parameters must conform to type restrictions on the formals, and must not allow aliasing. Actuals corresponding to var formals must be variable name expressions. Actual condition parameters must be forward conditions. They default to ROUTINEERROR. (Sections 5.9, 8.5.3).**
BEGIN [statements] ** [WHEN {is condi: statementsi}]** **END**	**The begin composition may be used to associate condition handlers with an arbitrary sequence of internal statements. (Section 8.3).**
SIGNAL condname	**A signal statement simply signals its forward condition. (Section 8.5).**
AWAIT ** [EACH i1: itype1,] ON stmt1** ** THEN statements1;** ** {[EACH in: itypen,] ON stmtn**	**The await composition provides a way of waiting concurrently on several events to occur. (Section 10.5.1).**

```
                    THEN statementsn}
  [WHEN {IS condi: statementsi}]
END
```

```
COBEGIN
  [EACH i1: itype1,]
        procname1 (exp11, .. exp1n);
  {[EACH in: itypen,]
        procnamen (expn1, .. expnn);}
  [WHEN {is condi: statementsi}]
END
```

A cobegin composition provides a way of running several processes concurrently. (Section 10.5.2).

Appendix F
Cross Reference of Operations by Type

This appendix lists the standard operations that are pre-defined for each Gypsy type. A special section under the buffer class lists functions on buffer histories. Where alternate notations for the same operation are available, they are presented side by side.

SIMPLE TYPES

Operations		Statements (s is a simple object)
EQ	=	s := exp;
NE		
LT	<	
LE		
GT	>	
GE		
MAX		
MIN		
INITIAL		
LOWER	(if the operand is a bounded type)	
UPPER	(if the operand is a bounded type)	

SCALAR TYPES

Operations		Statements (s is a scalar object)
EQ	=	s := exp;
NE		
LT	<	
LE		
GT	>	
GE		
MAX		
MIN		
INITIAL		
LOWER		
UPPER		
ORD		
SCALE		
PRED		
SUCC		

BOOLEAN TYPES

Operations			Statements	(b is a boolean object)
EQ	=	IFF	b := exp;	
NE				
LT	<			
LE				
GT	>			
GE				
MAX				
MIN				
INITIAL				
LOWER				
UPPER				
ORD				
SCALE				
PRED				
SUCC				
NOT				
AND				
OR				
IMP	->			
ALL				
SOME				

INTEGER TYPES

Operations			Statements	(i is an integer object)
EQ	=		i := exp;	
NE				
LT	<			
LE				
GT	>			
GE				
MAX				
MIN				
INITIAL				
LOWER		(if the operand is a subrange type)		
UPPER		(if the operand is a subrange type)		
DIV				
MOD				
+				
-		(unary)		
-		(binary)		
*				
/				
**				

RATIONAL TYPES

Operations		Statements (r is a rational object)
EQ	=	r := exp;
NE		
LT	<	
LE		
GT	>	
GE		
MAX		
MIN		
INITIAL		
+		
-	(unary)	
-	(binary)	
*		
/		
**		

ARRAY TYPES

Operations	Statements (a is an array object)
EQ =	a := exp;
NE	a[i] := exp;
a[i]	
a WITH ([i] := exp)	
INITIAL	

RECORD TYPES

Operations	Statements (r is a record object)
EQ =	r := exp;
NE	r.fieldname := exp;
r.fieldname	
r WITH (.fieldname := exp)	
INITIAL	

SET TYPES

Operations	Statements (s is a set object)
EQ =	s := exp;
NE	NEW exp INTO SET s;
ADJOIN	MOVE exp INTO SET s;
DIFFERENCE	MOVE exp1 FROM SET s TO exp2;
IN	MOVE exp1 FROM SET s INTO exp2;
INITIAL	REMOVE exp FROM SET s;
INTERSECT	GIVE exp FROM SET s TO b;
NULL	
OMIT	
SIZE	
SUB	
UNION	
(SET: e1, e2, ... en)	
(SET: e1 .. en)	

SEQUENCE TYPES

Operations	Statements (s is a sequence object)
EQ =	s := exp;
NE	s[i] := exp;
s[i]	MOVE exp BEFORE s[i];
s[i..j]	MOVE exp BEFORE SEQ s;
<:	MOVE exp BEHIND s[i];
:>	MOVE exp BEHIND SEQ s;
APPEND @	MOVE exp FROM SET setexp BEFORE s[i];
FIRST	MOVE exp FROM SET setexp BEFORE SEQ s;
IN	MOVE exp FROM SET setexp BEHIND s[i];
INITIAL	MOVE exp FROM SET setexp BEHIND SEQ s;
LAST	NEW exp BEFORE s[i];
NONFIRST	NEW exp BEFORE SEQ s;
NONLAST	NEW exp BEHIND s[i];
NULL	NEW exp BEHIND SEQ s;
SIZE	REMOVE s[i];
SUB	GIVE s[i] TO b;
(SEQ: e1, e2, ... en)	
(SEQ: e1 .. en)	
s WITH ([i] := exp)	
s WITH (BEFORE [i] := x)	
s WITH (BEHIND [i] := x)	

MAPPING TYPES

Operations

```
----------
 EQ    =
 NE
 m[x]
 INITIAL
 DOMAIN
 RANGE
 NULL
 DIFFERENCE
 INTERSECTION
 UNION
 SIZE
 SUB
 m WITH ([x] := y)
 m WITH (INTO [x] := y)
```

Statements (m is a mapping object)

```
----------
m := exp;
MOVE exp INTO m[i];
MOVE exp FROM SET setexp INTO m[i];
NEW exp INTO m[i];
REMOVE m[i];
GIVE m[i] TO b;
```

BUFFER TYPES

Operations	Statements (b is a buffer object)
FULL	SEND exp TO b;
EMPTY	RECEIVE exp FROM SET b;
CONTENT	GIVE exp TO b;
ALLFROM	GIVE exp FROM SET setexp TO b;
ALLTO	
INFROM	
INFROMMERGE	
OUTTO	
OUTTOMERGE	
TIMEDALLFROM	
TIMEDALLTO	
TIMEDINFROM	
TIMEDINFROMMERGE	
TIMEDOUTTO	
TIMEDOUTTOMERGE	
MESSAGES	
INITIAL	

The following are functions on buffer histories:

TIMEDMERGE
TIMEDORDER

ACTIVATIONID TYPES

Operations
EQ =

Index

Table of Contents

List of Figures

List of Tables