Report on the Language Gypsy

Version 1.0

Allen L. Ambler
Donald I. Good
Wilhelm F. Burger

August 1976 ICSCA-CMP-1

Certifiable Minicomputer Project
Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas 78712

Acknowledgments

The development of Gypsy has benefited greatly from the sustained collective efforts of many people with a diversity of experience and interests. These individuals have contributed significantly to the design and definition of Gypsy while enduring the seemingly endless procession of drafts of this report. Without their patient reading and continually probing analysis countless problems might have gone unnoticed. The authors are extremely grateful for the invaluable contributions of J.C. Browne, R.M. Cohen, C.G. Hoch, J.H. Howard, M.S. Moriconi, and R.E. Wells. A special thanks to our secretary M.D. Maier who entered and edited major portions of this Report.

Table of Contents

Table of Figures

Chapter 1

INTRODUCTION

Gypsy is the unifying element of a complete methodology for the construction of rigorously verified systems programs. This methodology provides an integrated way of specifying, designing, and implementing programs and of verifying that they always execute in conformity with their stated specifications, even in imperfect execution environments. The Gypsy language provides a precise means of expressing both a program and its specifications from initial specification through program design, implementation, verification, and successive modification and evolution. This integration of programming and specification facilities into a common language is the most significant single characteristic of Gypsy. This requires the program designer and verifier to comprehend only a single syntax and semantics throughout program development. This also allows program proofs to be constructed rigorously throughout all stages of development, thereby bringing maximal benefit to the total programming process.

The incorporation of specifications and programming facilities into a single language provides three complementary approaches to program verification. First, formal proofs that the program meets specifications can be constructed before any execution of the program occurs. Second, specifications can be validated by actual evaluation at run-time. Third, trace facilities provide a convenient mechanism for post-execution analysis if desired. This establishes a very effective relationship between program proof and run-time validation of specifications. Those specifications that are validated at run-time need not be proved, but can be assumed in the proofs, thus reducing significantly the size and complexity of the formal proofs. This, of course, increases program execution time, but provides the program designer and verifier with a choice of implementation and verification strategy.

One of the common, and often unstated, assumptions of formal program proofs is that of a perfect execution environment. For example, the problems of arithmetic on a finite set of integers often are ignored. Also it invariably is assumed that if an assignment x:=a is executed, then a successive reference to the value of x will equal a. In reality, this normally would imply that a computer memory never will drop a bit in the word(s) where x is stored. In Gypsy the conceptual span of both specifications and program code has been extended to include

program execution in imperfect environments. Specifications and program code concerning data integrity, error monitoring, and error isolation and recovery are expressed in a unified form along with the error-free environment statements.

In theory, methods of program proof can be applied to programs of any size. In practice, however, the provability of a program depends directly on the degree to which a program can be decomposed into small, independently provable units. Gypsy supports this kind of decomposition through facilities for both operational and data abstraction. The data abstraction is provided through a general mechanism for access control. These abstraction facilities can be applied uniformly either to programs or to specifications, and provide an effective basis for incremental design and verification of a complete program. Gypsy further supports this process by providing explicit facilities for top-down development. This permits higher-level units to be designed and verified even though some of the lower level details of their implementations may still be pending.

The original target of Gypsy was the expression of verifiable programs for communications processing such as those that might be found at the node of a computer network. This led to the incorporation of verifiable features for expressing concurrency and process synchronization and for expressing real-time dependencies. The result has been a high-level language for the development of general systems programs that can be verified to execute in conformity with precisely stated specifications.

1.1 Design of Gypsy

Gypsy was developed as an integrated programming and specification language to support the complete design, specification, coding and verification of systems software, with particular emphasis on communications software. Specific goals were:

* Complete Verifiability. Every feature in the language must be rigorously verifiable.

* Incremental Development. The language must support modular, incremental program development and verification. As best possible the language must simplify the verification process by encouraging small modules with tightly regulated interactions and by isolating and minimizing the effects of modifications to previously verified code. There must also be a facility for partial expression of program units.

* Systems Programming. The language must support the development of systems software. There must be facilities for expressing process concurrency and synchronizing process communication. There must also be facilities for expressing real-time dependencies.

* Imperfect Execution Environments. The language must support execution in imperfect environments. It must be possible to detect, isolate, and recover from run-time anomalies as well as monitor the program state.

* Specification Capability. The language must provide an extensive specification capability. For every property that is to be verified, there must be an adequate means of expressing it directly in the language. The integration of formal proof, run-time validation, and monitoring must be consistent and provide a complete whole.

The design of Gypsy has been a combined process of synthesis and contraction. Starting from Pascal [13] each existing Pascal construct was carefully analyzed and those which inhibited verification were modified or removed. The hierarchical definition structure was eliminated and protection lists were added to provide a tighter, more flexible environment for incremental program development and verification. Facilities for expressing concurrency, communication, synchronization, timing constraints, external events, error recovery, and monitoring were added, paying close attention to the requirements of the verification methodology. Each construct in the program code and the specification statements was designed to support the verification methodology. The program code syntax was modified to integrate these specification statements into a logically consistent and hopefully understandable language.

1.2 Designing for Verification

A language which is to facilitate coding and specification must not only include capabilities necessary for expressing the problem domain of interest, but must exclude language constructs whose semantics defeat, or impede, verification. We defer a discussion of Gypsy's specification statements until a later section for pedagogical reasons. Their development was, however, closely interwoven with that of the coding statements.

Verification of program code has only recently become a prominent factor in programming language design. While Pascal was influenced by verification considerations [5], more

recently Nucleus [10] and Alphard [23] have been expressly designed for verification by formal proofs. Gypsy also is specifically designed for verification, but verification by run-time validation as well as by formal proof. The first phase in the design of Gypsy was to develop a "conventional" language which was free from concepts known to render formal proof verification difficult. To this end, Pascal [13] was selected as a model and Gypsy was patterned after Pascal, but with significant differences.

Routines in Pascal can be nested to arbitrary depths which creates a hierarchy of nested "non-local" variables. Routines in Gypsy may not be nested and variables can only be defined within routines; hence, Gypsy has no non-local variables. This simplifies verification as well as incremental program development, which will be discussed in the next section.

Functions in Pascal can take either variable or value parameters and can only return values of a simple type. In Gypsy, functions are allowed constant and value parameters and they can return values of any non-buffer type. The restriction to these kinds of parameters, together with the absence of non-local variables, guarantees that functions produce no side-effects. This simplifies verification considerably.

Pascal allows routines to be included as parameters to other routines; Gypsy does not. This decision was made not because of the necessity to do dynamic type checking, but because the extra burden on the verification process did not appear worth the extra capability.

Certain of Pascal's data types do not appear at all in Gypsy. These are types "real", "class", "pointer", and "file".

Pascal has "if", "case", "for", "while", "repeat", and "goto" statements for execution control. Gypsy has a similar set of statements, "if", "case", "loop", and "leave", modified for proper placement of assertions and to eliminate the need for bracketing "begin-end" pairs. The "if" statement is conventional except for a trailing "end". The "case" statement has an additional keyword "is" and an optional "else" clause. The "loop" statement subsumes both the "while" and "repeat" constructs as well as the so-called "loop-and-a-half" construct and infinite loops. Termination and looping are controlled by "leave" statements. Gypsy has no "goto" statement.

1.3 Designing for Incremental Development

A language that is to support the development and evolution of verified programs also must consider the practical aspects of verification. In developing a verified program of any significant size, it is necessary that the program be written as a large collection of small, independently verifiable units. Otherwise, a formal proof easily can expand into a mass of detail and become unmanageable. Also for proofs to be maximally effective they should be carried out on a unit-by-unit basis as the program is developed. Further, it is the nature of systems programs that they are continuously undergoing evolution and with each modification some amount of reverification is necessary. It is, therefore essential that the amount of reverification be kept to a minimum. For these reasons, we sought language features which supported unit-by-unit manipulation, increased unit independence, and isolated unit interactions.

A Gypsy program consists of a series of "routine", "macro", "constant", and "type" units; which may appear in any order. If a reference can not be resolved locally within a particular unit, a search of the other external unit names is made. When an unresolved local reference is found to be an external unit name, then the appropriate information is extracted and the analysis continued. Access rights to any unit may be stated in an "access list." These access lists will be checked during the process of resolving references. The combination of units and access lists provides a high degree of code independence, plus a tightly controlled environment.

A routine is a "function", a "procedure", a "process", or a "program". A "program" unit defines the initial program execution point. Routine declarations can only appear at the unit level; hence, Gypsy does not provide a nested hierarchy. Besides favoring unit independence, it was felt (1) that a hierarchical structure failed to provide adequate program protection without access lists and (2) that with access lists and without nonlocal variables a hierarchical structure was unnecessary.

A macro unit binds a parameterized expression to a name. While macro expansions can be nested, they may not be recursively expanded as there would be no way to terminate a recursive expansion.

A constant unit parallels Pascal's constant declarations except that a constant may be of any non-buffer type including a structured type. This provides the means for referencing global values without allowing global variables or requiring them to be passed as parameters if they are not to be modified.

A type unit declares a new type either by itemizing its value set or by composing existing types. A type unit which includes an access list is the equivalent of an abstract data structure [7] [15] [23] [3] [9]. The intent of an abstract data type is to be able to construct a new type and to restrict access to the components of that type to operations representative of the type. It should be possible with a proper implementation to alter the implementation of the abstract type and the corresponding operations without impacting the program which employs the abstract type.

1.4 Designing for Concurrency and Real-Time

Programming languages have traditionally avoided concurrency; there have, however, been exceptions. The Burroughs family of extended Algol languages [16] provide processes and process communication, Bliss [21] provides coroutines and processes, Concurrent Pascal [3] combines processes and monitors, and Algol 68 [20] provides collateral elaboration of clauses. Several other languages have primitive means of accessing operating system functions which provide concurrency. Operating system research has generated a large number of concurrency and synchronization techniques which we will not attempt to reference. Two systems, RC4000 [1] and HYDRA [22], which were significant factors in our decision on how to specify and implement concurrency.

Gypsy has a routine type called a "process". It differs from a "procedure" only in the types of parameters allowed and in the manner of its invocation. Processes communicate only through message buffers [2]. A message "buffer" is a finite length queue on which there are only two operations defined, "send"(enqueue) and "receive"(dequeue). The queue is manipulated by a strict FCFS algorithm. Whenever a "send" is made on a full buffer the sending process is suspended until the condition is remedied. Likewise, a "receive" on an empty buffer will cause the process to be suspended. Associated with every buffer is a semaphore which guarantees mutually exclusive access to the buffer.

Concurrent processes are initiated by a "cobegin..end" statement and may or may not terminate. Only when all processes called within a "cobegin" statement terminate will the statement following the "cobegin" be executed.

Polling is an important function of real-time systems; hence, it must be possible to poll a buffer without being

suspended indefinitely trying to receive from an empty buffer. Gypsy has an "await" statement which allows the simultaneous waiting on the completion of any one of several buffer operations. An "await" is in many respects a guarded command [8], except that it has a very restricted set of guards and it has an optional time-out clause. The time-out clause specifies what is to be done if none of the requested operations completes by a certain time.

The concept of (real) time is provided by "clock" variables. A clock variable is a special variable which may not be modified by the program, but which is always changing. There may be any number of clocks in a program, but there is no guarantee that they will be synchronized. Gypsy programs may be distributed across many machines and this makes synchronization virtually impossible.

1.5 Designing for Imperfect Execution Environments

An attribute of real-time software often overlooked in programming languages is the existence of both hardware and software faults. Fault detection, isolation, and recovery is an essential function in real-time software and consequently, languages for expressing such software should (1) provide capabilities for fault control programming and (2) provide an interface to the hardware which allows for the detection, isolation, and recovery of faults. The work of the Newcastle group [18] represents virtually all of the previous efforts on this topic.

A "condition" in Gypsy is an instantaneous event which may occur during the execution of a program. There is a large class of predefined "conditions" which correspond to hardware errors and dynamic language semantics errors, such as "caseerror". Programmers may, in addition, name and signal fault conditions by using a "signal" statement or an "otherwise" clause on a specification (discussed in the next section).

Any statement ending with the word "end", may optionally end with a "condition clause" followed by the word "end". The effect of the condition clause is that whenever a condition occurs, an immediate branch is taken to the condition clause, of the innermost containing statement, which specifies an action for that condition. Searching for the innermost condition clause may involve exiting a routine. After the condition clause is executed, control does not return to where it was before the fault, but instead drops out of the statement whose condition

clause was executed. In some sense, a condition clause is a restricted version of a PL/I "on" condition which resembles one of Zahn's event driven case statements [24].

1.6 Designing for Specification

Gypsy plays the dual role of programming and specification language. The specification component of the language permits the precise expression of desired functional properties of key parts of the program. These properties are stated in terms of valid states that are to be maintained on the data objects of the program at various points in the program computation. The objective of a verification is to show that the computation always proceeds in conformity with the stated specifications. The conformity of the program with its specification can in most cases be either proved prior to execution or validated during execution. The same specification methods are used in both approaches to verification.

All specifications in Gypsy are stated as boolean-valued expressions. These specifications are designated to be verified either by proof, by run-time validation, or simply assumed. Specifications that are proved or assumed need not be evaluated at run-time, and therefore, they are permitted to contain special operations and types that could not otherwise be permitted. For example, boolean expressions may contain the logical quantifiers "for all" and "there exists" and refer to rational numbers and infinite sequences.

The most familiar kinds of specifications used in Gypsy are the "entry", "exit", and "assert" statements for procedures and functions. These follow the same form as that introduced by [12] for proving Pascal programs. The "exit" specification is interpreted in the weak sense, i.e. it holds if the program terminates.

"Entry", "exit", and "assert" specifications also can be used with processes. However, processes often are intentionally programmed never to terminate, and therefore an "exit" specification may be of no value. Specifications can be stated for non-terminating processes through "block" specifications. A "block" specification holds whenever the process is suspended by a buffer operation. This provides a temporary halting point.

Specifications for routines performing buffer manipulations normally are stated in terms of effects on buffer histories. In the terminology of [6], these are "mythical variables", but

they are provided in a uniform way rather than being installed by the programmer. Associated with every buffer b are several histories that are relevant to specifications and to the proof methodology. For example, "b.infrom" refers to the sequence of objects received "in" from the buffer by the process, and "b.outto" is the sequence of objects sent "out" to the buffer from the process.

Any sequence of "var" declarations can be followed by a "keep" specification. The "keep" expression must be maintained throughout the immediate scope of the "var" declaration. A procedure or function call releases the "keep", but the called unit must reestablish it before returning. This type of assertion is similar to those used by [19] for run-time validation.

Routines that have access to the internal structure of a type have both internal and external specifications. This follows the specification methods of Alphard forms [23]. External specifications are visible to the outside environment; internal ones are not. The external specifications are the "entry", "block", and "exit" specifications. "Centry", "cblock", and "cexit" are the corresponding internal specifications. The internal specifications may refer to the internal (concrete) structure of accessible types; the external specifications may not.

Two kinds of specifications can be stated for Gypsy type definitions, "require" and "axiom". The require specification follows Alphard and is a precondition on the type parameters that is necessary for the proper creation of an object of that type. The axiom is a relation among the functions that have access to the type.

This set of specification methods provides powerful mechanisms for stating functional properties of programs, and formal proof methods have been defined for proving each of these types of properties. The specifications do not, at this time, directly permit the definition of quantitative aspects of program behavior such as resource utilization.

Chapter 2

BASIC LANGUAGE

Gypsy's basic programming language without extensions for concurrency, verification, or error handling bears a strong resemblance to Pascal. There is of course good reason for the similarity between these two languages; in designing Gypsy every effort was made to exploit reliable, consistent, and familiar features of existing languages and in particular of Pascal. Pascal provided a good, relatively clean starting point, and as nearly as was possible, its design has been preserved, but coerced into a framework consistent with the goals of reliable, secure systems software.

In this chapter of the report the basic programming features are described. In the succeeding parts of this report extensions for concurrency, verification, and error handling are discussed.

2.1 Environment

It will be helpful if we establish at the outset a basic understanding of the program environment for Gypsy programs, especially since it is unlike conventional program environments. Gypsy programs are composed of "units" which are stored in an on-line data base. These units are separate, but not independent, modules of code. They may be entered, parsed, proven, etc. in any order subject only to interdependency constraints. For instance, a procedure A might be entered at time t0, parsed at time t1, proven at time t2, assembled at time t3, and executed at time t4; however, if A calls a function B then the following interdependencies arise: (1) the function header (not necessarily the entire function) for B must be entered before time t1, (2) the external specifications for B must be entered before t2, and (3) the function body for B must be entered and the entire function parsed and assembled before t4. The process of compilation will be described more in section 2.11. The various times t0, t1, t2, t3, and t4 give rise to the following terminology "entry-time", "parse-time", "proof-time", "assembly-time", and "run-time", respectively.

Various portions of a Gypsy program are evaluated at different "times". Consequently, there are different constraints

on essentially identical syntactic constructs depending upon the evaluation time. For instance, an expression might be evaluated at parse-time, at proof-time, or at run-time. If it is to be evaluated at parse-time, then it cannot contain operands which do not yield parse-time values, i.e. variables. If it is to be evaluated at proof-time and never to be actually executed, then an expression is allowed to contain operations such as existential quantifiers which would otherwise be forbidden. For parse-time evaluation, all operations which cannot be reasonably implemented are disallowed. As various parts of the language are defined, these distinctions will be discussed with more particulars, but for now the essential concept of different evaluation times should be understood.

2.2 Notation

Standard Backus-Naur form uses angular brackets "<" and ">" to enclose strings of characters representing non-terminals; a double colon equal sign "::=" as the rewriting operator; and a vertical bar "|" to denote alternative right hand sides. Strings of symbols not enclosed in angular brackets represent themselves and are considered as terminals. The notation used here departs from BNF by employing curved brackets "{" and "}" along with a postscript +, *, or ! to mean, respectively, one or more repetitions, zero or more repetitions, or zero or one repetitions of the enclosed phrase.

Comments, denoted by double quotes and allowed within non-terminal names, impart semantic meaning only and may be ignored for syntactic form. These "meta-comments" are interpreted by substituting possible values for the quoted string uniformly in the production to derive an "instantiation" of the production; for example, for

<assignment> ::= <"mode" variable> := <"mode" expression>

one could substitute various modes, such as integer, boolean, etc. to derive

<assignment> ::= <integer variable> := <integer expression>
<assignment> ::= <boolean variable> := <boolean expression>
etc.

The backslash "\" has been used in several productions to separate two or more terminal symbols which may be used interchangeably. Only one of these terminals may be used in any

occurrence of the production, but it doesn't matter which one is chosen nor must the choice be consistent.

2.3 Character Set

The exact character set will no doubt be determined by the particular machines involved in implementing Gypsy; however, a minimum of the 7-bit ASCII standard character set is recommended. The character set is divided into subsets as follows:

```
<symbol> ::= <letter>|<digit>|<score>|<prime>|<opcode>|
    <quote>|<blank>|<bracket>|<special>
<letter> ::= # upper/lower case ASCII letter #
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<score> ::= _
<prime> ::= ~
<opcode> ::= +|-|*|/|<|>|@|=|.|,|;|:
<quote> ::= "
<blank> ::=
<bracket> ::= <|>|(|)|{|}
<special> ::= # all other ASCII characters #
```

Upper and lower case letters may be used interchangeably in constructing Gypsy programs. This allows effective use of cases for enhancing readability without generating typographical errors.

Square brackets "[" and "]" and parentheses brackets "(" and ")" may be used interchangeably subject only to the stipulation that matching brackets must be of the same kind, i.e. both square or both parentheses brackets. In the remainder of this report we will use only parentheses; however, the user is free to substitute square brackets in their place.

If Gypsy is implemented on machines supporting only lesser character sets it may be necessary to designate some alternative character sequences, for instance "(*" and "*)", as well as "{" and "}", to designate comments.

2.4 Tokens

The essential tokens are identifiers, numbers, characters, and strings. Identifiers must begin with a letter, but

thereafter they may be composed of any combination of letters and digits. Identifiers may be of any length and, to decrease compiler dependence, Gypsy compilers are required to differentiate complete identifiers, not just the first "N" characters as is frequently done.

Identifiers allow two conventions for the sake of readability. First, upper and lower case letters may be used interchangeably within names and second, the underscore character may be used freely within names to separate abbreviations. The underscore is, however, never allowed to either begin or end an identifier name. The underscore used by itself has a special meaning which is explained in section 2.5.6.

The set of numbers are the unsigned integers. While the syntax allows arbitrary length numbers, any implementation may have a maximum allowable value.

Strings are character sequences appearing between a pair of string markers which are usually double quotes, but which may be one or more alternate special characters specified unique to the particular implementation. Within a string a pair of consecutive markers will be interpreted as a single character of the string and not the string terminator. This allows for the insertion of quotes in quoted strings.

Single characters are indicated by a single quote placed immediately before the desired character. By using a different notation for a single character than for a string containing one character, any possible type ambiguity is avoided.

Strings of characters enclosed in comment markers, "{" and "}", will be interpreted as comments. All comments are removed during compilation and have no effect on the resulting compilation or verification. Comments may appear in any context, except within tokens.

```
<token> ::= <id>|<number>|<character>|<string>|<comment>
<id> ::= <letter> {{<letterdigitscore>}* <letterdigit>}!
<number> ::= {<digit>}+
<character> ::= <prime> <symbol>
<string> ::= <quote> {<symbol>}* <quote>
<letterdigitscore> ::= <letterdigit>|<score>
<letterdigit> ::= <letter>|<digit>
<comment> ::= { #string not containing { or }# }
```

Example:

```
SymbolTable    Hash_Table    100    "string"    x'    'x
```

2.5 Types

Type units consist of a type header followed by a type body. In the succeeding sections of this chapter, simple and compound types will be discussed; more complex type declarations will be postponed until Chapter 3. In the syntax given below there are several forward references, such as access list (section 2.11.2), type parameters (section 3.1.1), pending (section 3.4), and type specifications (section 3.6.1). For the present these can be ignored as they are not necessary for simple programs.

```
<type unit> ::= <"type" type header> =
    <"type" type body>|pending
<"type" type header> ::= type {<access list>}!
    <"type type name" id> {<formal type parameter list>}!
    {<access list>}!
<"type" type body> ::= <"type" type declaration>|
    begin {<type spec> ;}* <"type" type declaration> end|
    begin {<type spec> ;}* pending end
```

2.5.1 Simple Types

The simple types in Gypsy consist of the scalar types, which include boolean, plus two special predefined simple types: rational and character. Type rational is distinguished for two reasons (1) its value set is composed of numbers rather than identifiers and (2) its value set is by definition unbounded. Type character is distinguished for only the former reason.

A scalar type is defined by a parenthesized list of identifiers. Each identifier in the list is defined to be a value of the new type and whenever the identifier appears subsequently it will be recognized as a value of the newly defined type. The order in which the identifiers appear within the list is significant and is interpreted as the ordering relation for the new type. For example,

```
type DaysOfWeek = (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
```

defines a new scalar type named DaysOfWeek which has seven values Mon, Tues, etc. and for which there is an implicit ordering such that Tues < Wed, etc..

In defining a new scalar type all of the identifiers appearing in the declaration must be unique over the scope of any instantiation of the type. Thus, two types may have conflicting

value names provided that there never occurs instantiations of both types within any scope. The scope of declarations will be discussed more carefully in section 2.11.1.

The type boolean which consists of only two values is predefined by the following scalar definition:

```
type boolean = (false, true);
```

The reader is reminded that ordering is significant.

Type rational is precisely the rational numbers (i.e., numbers of the form "n/m", where "n" and "m" are "integers" and "m" is not zero). Type integer is defined to be the numeric values ..., -3, -2, -1, 0, 1, 2, 3, ... and is by definition unbounded. In Gypsy the integers are conceived as special rational numbers which have the property that "m=1". This "special" relationship will be discussed again in section 2.5.3.

Type character can be defined as a scalar type, and indeed is, but because of the special treatment of its value set it will be distinguished and not referred to as a scalar type. For example:

```
type character = (BREAK, SOH, ..., ~, DEL); i.e. the
    standard 7-bit ASCII character code set.
```

Unfortunately, the character code set is machine dependent.

```
<"simple type" id> ::= rational|integer|character|
    <"scalar type" id>
<"scalar type" id> ::= boolean|<"scalar type name" id>
<"scalar" type declaration> ::= ( <"scalar value" id>
    {, <"scalar value" id>}* )
```

Example:

```
type MonthsOfYear = (Jan, Feb, Mar, Apr, May, Jun, Jul,
    Aug, Sep, Oct, Nov, Dec);
type Color = (Red, Blue, Yellow);
type Politician = (Rep, Sen, VPres, Pres);
```

2.5.2 Simple Values

The simple values are precisely the numbers, the characters, and the scalar values which were defined by scalar type declarations.

Every type possible in Gypsy is composed from the simple types; hence the set of possible values in Gypsy programs consists of all possible legal compositions of the simple values. A discussion of the possible compositions is deferred to section 2.5.6.

```
<"simple" value> ::= <"rational" value>|
    <"character" value>|<"scalar" value>
<"rational" value> ::= <"integer" value>|
    <"integer" value> / <"integer" value>
<"integer" value> ::= <number>|- <number>
<"character" value> ::= <character>|<"character value" id>
<"scalar" value> ::= <"scalar value" id>
```

Example:

```
2/3    1001    "a"    """"    BREAK    Mon    Pres
```

2.5.3 Modes and Restrictions

In Gypsy a type is composed of a mode, or basic value set, and a possibly empty set of restrictions upon that mode, which serves to define a subset of the basic value set. All scalar types, as well as all predefined types, are considered initially unrestricted; hence, the simple types defined in section 2.5.1 are all unrestricted and of mode simple. Note that unrestricted is not synonymous with infinite.

A type restriction is a means of defining a type from an existing mode by restricting the mode's value set. Type integer is defined as a special restriction of mode rational which requires that the divisor portion of the rational be precisely one and thus be omitted. Hence, type integer is of mode rational, but restricted to those values which are "whole" numbers. This kind of restriction is unique to the integers.

For simple types the most prevalent means of defining a restricted value set is by a range restriction. This is accomplished by specifying a subrange of the existing value set. While range restrictions may be advantageous for any simple mode, they are required of all run-time integer variables (proof-time integer variables need not be restricted).

A range restriction is appended to the simple type name and is specified by a pair of expressions in brackets, for example:

```
type byteint = integer(-128 .. 128);
```

Both expressions must evaluate to values of the prescribed type with the first value (left) being less than or equal to the second value (right).

There are a number of implications for range restrictions. It is assumed that whenever the compiler cannot guarantee that values will stay within that range (which is generally the case), then code will be generated to perform the necessary checks during execution. If the value cannot be stored conveniently (for instance, in a single word) such that the storage unit is large enough to satisfy the specified value set, then this condition will be reported as an error. Conversely, it is assumed that the value will be stored in as small a storage unit as is both possible and consistent with reasonable referencing efficiency.

The range expression pair need not necessarily evaluate at parse-time; however, their modes must be decidable. When the range is unknown at parse-time the compiler will generate storage for the worst case it can handle. If the run-time evaluation results exceed the worst case then a run-time error will result.

In the case of the two predefined "special" simple types, rational and character, whose value sets are uniquely distinguishable and for which the mode can be deduced from the mode of the expressions, the mode identifier only supplies redundant information and may be omitted if desired. Whenever there is any ambiguity over the modes of the expressions, then the mode must be explicitly supplied.

A type "int" is predefined to be the largest range of integer values representable as a single addressable object on a particular machine, for example on the DEC-1Ø int would be defined as:

```
type int = integer(-2**35..2**35-1);
```

```
<"simple" type declaration> ::= {<"simple type name" id>}!
    <"simple" range>
<"simple" range> ::= ( <"simple" expression> ..
    <"simple" expression> )
```

Example:

```
type Summer = MonthsOfYear(Jun .. Aug);
type Weekday = DaysOfWeek(Mon .. Fri);
type Letters = ("a" .. "z");
```

2.5.4 Type References

A type reference is a reference to a previously defined mode possibly accompanied by some additional restrictions. In the simple case a type reference is simply the name of a type. The alternatives are the name of a simple type followed by a restriction, the restriction only (if the type can be deduced, see section 2.5.3), or the name of a parameterized type with an expression list specifying the actual restrictions (see section 3.1.1).

```
<"type" type reference> ::= <"type type name" id>|
    <"type type name" id> <"type" range>|<"type" range>|
    <"type type name" id> <actual type parameter list>
<"type" formal type reference> ::=
    <"type" type reference>|<"type type name" id>
    <actual/formal type parameter list>
<actual type parameter list> ::= ( <actual type parameter>
    {, <actual type parameter>}* )|{empty}
<actual type parameter> ::= <expression>|{empty}
```

2.5.5 Compound Types

The compound types are records and arrays. Records allow the composition of dissimilar objects as fields of a larger object by specifically naming each component field. The advantage to a record definition as opposed to separate declarations of the individual fields is that the compound object now has a name and may be manipulated as a compound object as well as by its individual fields.

Field names need only be unique within individual records and may be reused in subsequent record definitions. The field names for a record mode become defined in any scope when there is declared an instantiation of that mode in that scope, except under special circumstances described in section 2.11.2.

It is helpful at this point if a distinction is made between "base" names and "modifier" names. A base name refers to the entire object, while a modifier name qualifies a base reference to refer to a subcomponent of the entire object. Base names may appear by themselves, but modifier names are never allowed to appear except in conjunction with their corresponding base names. Given these definitions there is one rule regarding name uniqueness: All base names used within any routine must be unique.

What names are base names? Base names are type names, constant names, scalar value names, variable names, parameter names, and routine names. Fields within records are modifier names. Record definitions have the same general form that simple type declarations have, namely a type header followed by a type body. Within the type body there are a number of local field declarations each itself composed of a field header followed by a type declaration.

In allocating storage for records, fields will be packed as tightly as is consistent with efficient referencing and without breaking fields across unnatural boundaries. The compiler will be left the ultimate decision as to the organization of all storage allocations.

```
<"record of typel...typen" type declaration> ::=
    record ( <"typel" field declaration>
    {; <"typei" field declaration>}* )
<"type" field declaration> ::= <"type" field header>
    : <"type" type declaration>
<"type" field header> ::= <"type field name" id>
    {, <"type field name" id>}*
```

Example:

```
type Date = record (
        Month : MonthsOfYear;
        Day : (1..31);
        Year : (1900..2000) );

type TaxRecord = record (
        SocSecNo : integer(0..999999999);
        Income, Tax : integer(-10**12..10**12);
        DateFiled : record(MM, DD, YY : (1..99)) );
```

Arrays provide for the repetition of identically typed objects by providing an index value set from which each member is associated with an item of the named type. This provides a means of referencing each item within the array. Gypsy allows only one dimensional arrays; however, it is permissible to declare arrays of arrays of ... etc.

Type array declarations are declared in an analogous manner to previous type declarations. The index set is indicated by a type reference and the array will contain one element for every value of the index type set. Recalling that the integers are infinite, this means that an array declaration could be potentially infinite. Arrays which are to be allocated during run-time must necessarily be bounded. This is accomplished by bounding the index type (assuming it is not already bounded).

Storage for arrays, like records, will be allocated as concisely as is consistent with efficient referencing. Since the normal means of referencing arrays is by indexing, attention will be paid to allowing efficient index computation.

```
<"array typel of type2" type declaration> ::=
    array ( <"typel" type reference> ) of
    <"type2" type declaration>
```

Example:

```
type Name = array(integer(1..20)) of character;
type Matrix = array((1..N)) of array ((1..N)) of (1..N);
type CharTab = array(character) of character;
type Table = array ((0..63)) of integer(0..1000);
type TaxRecArray = array((1..N)) of TaxRecord;
```

2.5.6 Value Compositions

Values of every definable type (except buffers, see section 3.2.1) may be constructed from simple values (defined in section 2.5.2). Simple types as well as compound types may be composed; furthermore, in most contexts the type name is unnecessary and may be omitted. Value composition will be particularly useful in variable declarations (see section 2.7).

The set of values for any compound type is precisely the set of compound values composed by selecting every possible combination of component values, i.e. the Cartesian or cross product of the component values. To construct values for the compound types the type name is used as an operator for composing components of the correct component types. The format of a compound value composition is the type name followed by a bracketed list of the component values in order and separated by commas. For example,

```
Date(Jan,5,1976)
```

composes a record of type "Date" from the component expressions "Jan", "5", and "1976" which are of the correct corresponding types "MonthsOfYear", "(1..31)", and "(1900:2000)". Similarly,

```
Name("G","Y","P","S","Y"," "," "," "," "," "," ",
     " "," "," "," "," "," "," "," "," "," "," ")
```

creates a value of type "Name" from twenty components of type "character".

The last example illustrates the need for a shorthand notation for expressing a repeated value. This need is met by the pseudo operator "*:". The expression x *: y is interpreted as x repetitions of the expression y with each adjacent pair separated by a comma, i.e. y,y,y,...y. Thus the same example could be written

```
Name("G","Y","P","S","Y",15*:" ")
```

For types composed of unnamed types (arrays of arrays of ... or records of records of ...) a composition will take a corresponding form; in particular, it will have an unnamed type composition within the named type composition. The unnamed composition is allowed only because its type is discerned from its position within the encompassing type composition. An unnamed composition is indicated by a "_" (underscore) appearing immediately before a bracketed expression list which is contained within the named type composition.

In order that modes may be checked completely at parse-time the "repetition" expression must evaluate at parse-time.

```
<"simple" composition> ::= <"simple type name" id>
    ( <"simple" expression> )|_ ( <"simple" expression> )
<"record of type1...typen" composition> ::=
    <"record of type1...typen type name" id> (
    <"type1" Fomponent> {, <"typei" component>}* )|
    _ ( <"type1" component> {, <"typei" component>}* )
<"array type1 of type2" composition> ::=
    <"array type1 of type2 type name" id> (
    <"type2" component> {, <"type2" component>}* )|
    _ ( <"type2" component> {, <"type2" component>}* )
<"type" component> ::= {<"integer" expression> *:}!
    <"type" expression>
```

Example:

```
Matrix(N *: _(N *: Ø))
TaxRecord(123456789, 43750, 13773, _(4, 15, 1976))
```

2.5.7 Mode Equivalence

Two modes are said to be equivalent if and only if after recursively substituting all user defined modes for their corresponding types and all formal type parameter types for their corresponding formal parameter names, then the resulting modes are identical. For example, the following pairs are mode equivalent:

```
array ((1..5)) of int   and   array ((1..10)) of int
Date   and   record(x:MonthsOfYear; y,z:int)
```

2.6 Constants

A constant is a name which is bound to a particular value in its declaration and which retains that value throughout the scope of the declaration. Again, a complete discussion of scope is deferred until section 2.11.1.

A constant declaration consists of an identifier followed by a constant expression. Whenever the mode of the expression is not explicitly provided (as is the case with a value composition or a function call) and the mode of the expression is not "rational", "character", "boolean", or "string" (section 3.1.2), then the optional type reference is required.

In Gypsy, constant declarations can occur either as units themselves or as local declarations within routine units. Only when they appear as units is it permissible to associate an access list with the constant and only constant units are required to evaluate at parse-time. Local constant declarations need not necessarily evaluate prior to run-time.

Constant units may be entered and referenced in any order; however, they are not allowed to recursively reference each other.

```
<constant unit> ::= <"type" constant header>
    {: <"type" type reference>}! = <"type" expression>|
    <"type" constant header> : <"type" type reference>
    = pending
<local constant> ::= const <"type constant name" id>
    {: <"type" type reference>}! = <"type" expression>|
    const <"type constant name" id> :
    <"type" type reference> = pending
<"type" constant header> ::= const {<access list>}!
    <"type constant name" id>
```

Example:

```
const NDaysInYear : integer = 365;
const MaxHours = 40;
const Christmas = Date(Dec, 25, 1976);
```

2.7 Variables

A variable is an object of a prescribed type which may be assigned different values of the prescribed type arbitrarily often. A variable is declared by a variable declaration which consists of a list of identifiers followed by a type reference. The effect of the declaration is to allocate storage of sufficient size to hold the value set for the type indicated and to associate a name with that storage. Subsequent references to that variable are references to the value stored in the allocated storage of the prescribed type. A variable declaration is sometimes referred to as an instantiation of the specified type.

Any variable may be initialized at the time of its declaration by an initialization expression. This initialization will be performed every time that the variable is allocated. If the expression can be evaluated at parse-time (i.e. consists of only values and constants) then initialization is simple and efficient; otherwise, the compiler will generate code to evaluate the expression and perform the proper initialization.

Whenever the initialization expression provides redundant type information, i.e. the variable type (including restrictions) can be deduced from the expression (as is the case with a value composition or function call), then the type reference may be omitted. It is never the case that both the type reference and the variable initialization may be omitted.

```
<local variable> ::= <"type" variable header>
    {: <"type" type reference>}! {:= <"type" expression>}!|
    <"type" variable header> : <"type" type reference>
    = pending
<"type" variable header> ::= var <"type variable name" id>
    {, <"type variable name" id>}*
```

Example:

```
var Day : DaysOfWeek;
var i, j, k : integer(1..N) := 1;
var Switch : boolean := false;
var SyTab := Table(64*:0);
var TaxFile : TaxRecArray;
```

2.7.1 Variable Reference

A variable reference is a reference to a portion of the storage assigned to and associated with a variable name. The

syntax and semantics of the variable reference are related to the type specified in the variable declaration.

Syntactically, variable references are identical in all contexts; however, semantically, there is a distinction between evaluation and assignment references. In the former case the result of the reference is the value contained at the location associated with the reference; while in the latter case the result of the reference is the location itself. It is precisely this distinction that prevents function references from occurring in the context of an assignment reference.

Simple variables can only be referenced in their entirety; while, compound variables can be referenced both in their entirety and by their subcomponents.

Record components are referenced by following the record reference with the desired field name separated by a dot (period).

Array components are referenced by index expressions of the prescribed mode written in brackets immediately following the array reference. The value resulting from the evaluation of the index expression must satisfy the restrictions defined for the index type. The compiler will generate code whenever necessary to assure that the restrictions are not violated.

```
<"type" variable reference> ::= <"type variable name" id>|
    <"record of ...type..." variable reference> .
    <"type field name" id>|
    <"array typel of type" variable reference> (
    <"typel" expression> )
```

Example:

```
i
Day
Switch
SyTab
SyTab(i)
TaxFile(i+j).DateFiled.MM
```

2.8 Expressions

An expression consists of a simple value, a value composition, a constant, a variable reference, a function call, or an operational formula; each of which may itself be composed of expressions.

As was mentioned in section expressions may appear in several contexts and depending upon the context they will be evaluated at different times and will be allowed different operations. As the standard operations and functions are introduced in the following sections, those operations allowable only for proof-time evaluation will be pointed out.

In Gypsy, it is always possible to determine the mode of an expression from the modes of its components without additional context information. This means that Gypsy enforces rigid type matching and performs no coercions.

```
<"type" expression> ::= <"type" value>|
    <"type" composition>|<"type constant name" id>|
    <"type" variable reference>|
    <"type" primed variable reference>|
    <"type" function call> ( <"type" expression> )
```

2.8.1 Function Calls

The syntax and semantics of a function call are almost identical to that of a routine call with the obvious distinction that one appears as a statement and the other as an expression returning a value. A more complete discussion of functions, their parameters and their restrictions, will be given in later sections (2.10.1 and 2.10.3).

```
<"type" function call> ::= <"type function name" id>
    {<actual parameter list>}!
<actual parameter list> ::= ( <actual parameter>
    {, <actual parameter list>}* )
<actual parameter> ::= <expression>|<"condition name" id>
```

2.8.2 Standard Operations

In this section the standard predefined operations for simple and compound types are presented.

Operators are functions which are distinguished by their infix syntactic form as opposed to the conventional parenthesized functional form. Because of the lack of parentheses, an operator's parameters may appear ambiguously specified. In order to resolve this ambiguity, precedence relations are defined on all operators which effectively parenthesize any expression. These precedences are summarized in Appendix B.

Whenever it is desired that the evaluation of an expression involving operators be performed in a manner other than that specified by the precedence relations, the order can be indicated by explicit parenthesization of the expression. In addition it may be desirable in some cases to include extra parentheses to increase readability.

The rational operators are: negation, addition, subtraction, multiplication, truncated division, modulus, and exponentiation. In addition to the customary operator symbols Gypsy provides a corresponding set of reserved words which may be used interchangeably to suit personal taste.

```
<"rational" expression> ::= plus\+ <"rational" expression>|
    minus\- <"rational" expression>|
    <"rational" expression> power\**
    <"rational" expression>|<"rational" expression>
    plus\+ <"rational" expression>|
    <"rational" expression> minus\-
    <"rational" expression>|<"rational" expression>
    times\* <"rational" expression>|
    <"rational" expression> divide\/
    <"rational" expression>|<"integer" expression>
    div\// <"integer" expression>|
    <"integer" expression> modulus\mod
    <"integer" expression>
```

Example:

```
i + j ** 2 * k
i + ((j ** 2) * k)
i plus j power 2 times k
```

Each of the above expressions is equivalent.

The relational operators are defined for simple types only. The definition of the relations is defined on the implicit ordering of the value set induced by their definitions. In addition to the traditional relational operators eq, ne, lt, le, gt, and ge, there is an "in" operator which evaluates true if and only if the value of the left operand is within the range specified by the right operand.

The operations equal and not equal are the only standard operations which apply to compound types. Two arrays are equal if and only if they are of the same mode, restrictions, and all of their values are identical. Two records are equal if their modes are equivalent and each pair of corresponding components are equal.

```
<"boolean" expression> ::= <"simple" expression> eq\=
```

```
<"simple" expression>|<"simple" expression> ne\<>
<"simple" expression>|<"compound" expression> eq\=
<"compound" expression>|<"compound" expression> ne\<>
<"compound" expression>|<"simple" expression> lt\<
<"simple" expression>|<"simple" expression> le\<=
<"simple" expression>|<"simple" expression> gt\>
<"simple" expression>|<"simple" expression> ge\>=
<"simple" expression>|<"simple" expression> in
<"simple" range>
```

Example:

```
3 < 7
"a" < "z"
Wed >= Wed
false < true
"A" < "a"
5 in (1..1Ø)
```

All of the above relations are true.

The boolean operators are not, or, and, implication, and equivalence. Note that because of precedence relations the operators = and eqv are not equivalent.

```
<"boolean" expression> ::= not <"boolean" expression>|
    <"boolean" expression> or <"boolean" expression>|
    <"boolean" expression> and <"boolean" expression>|
    <"boolean" expression> imp\-> <"boolean" expression>|
    <"boolean" expression> iff\eqv <"boolean" expression>
```

Example:

```
Switch imp i < j
i < j iff j > i
```

The quantifier operators given in this section are allowed only in proof-time evaluations and will be of no use until specification statements are introduced in section 3.6. However we introduce them now just to hold their place in the precedence hierarchy. The two operators are the traditional logic existential and universal quantifiers. Both operators have a local variable definition as part of their semantic definition.

```
<"boolean" expression> ::= some <"type variable name" id>
    {, <"type variable name" id>}* :
    <"simple" type reference> , <"boolean" expression>|
    all <"type variable name" id>
    {, <"type variable name" id>}* :
    <"simple" type reference> , <"boolean" expression>
```

Example:

```
all x,y:integer(1..N), M(x)(y) = ∅
```

The above example translates: for all x and y, integers in the range 1 to N, M(x)(y) = ∅.

The conditional expression is fully bracketed and hence requires no operator precedence rule.

```
<"type" expression> ::= if <"boolean" expression>
    then <"type" expression> else <"type" expression> fi
```

Example:

```
if a < b then b else a fi
```

2.8.3 Standard Functions

There are two sets of predefined functions that are available in Gypsy. These are the order, bound, and conversion functions. The order functions, pred and succ, return the predecessor and successor values of the parameter value. The order functions, min and max, return the smaller and larger, respectively, of a pair of simple values.

```
<"simple" expression> ::= pred ( <"simple" expression ) |
    succ ( <"simple" expression> ) |
    min ( <"simple" expression> , <"simple" expression> ) |
    max ( <"simple" expression> , <"simple" expression> )
```

Example:

```
pred("b") = "a"
succ(Mon) = Tues
succ(succ(pred(2))) = 3
min(3,5) = max(1,3)
```

The bound functions, upper and lower, return the largest and smallest values of the named type.

```
<"simple" expression> ::= lower ( <"simple type name" id>
    ) |upper ( <"simple type name" id> )
```

Example:

```
upper(boolean) = true
```

The conversion functions, ord and scale, convert the values of any simple type to the corresponding integer values and vice versa. Since ord can deduce the type from the expression it does not require the type name in order to perform the conversion; whereas, scale requires the desired type name.

```
<"simple" expression> ::= scale ( <"integer" expression> ,
    <"simple" type reference> )
<"integer" expression> ::= ord ( <"simple" expression> )
```

Example:

```
scale(0,DaysOfWeek) = Mon
ord(false) = 0
```

2.9 Statements

Statements in Gypsy are either simple or compound. Compound statements are statements which are structured compositions of other statements.

```
<statement list> ::= {<statement> ;}* {<statement>}!|
    pending|pending ;|;
<statement> ::= <simple statement>|<compound statement>
<simple statement> ::= <assignment statement>|
    <routine call statement>|<leave statement>|
    <buffer statement>|<clock statement>|
    <statement spec>|<signal statement>
<compound statement> ::= <if statement>|
    <case statement>|<loop statement>|
    <begin statement>|<cobegin statement>|<await statement>
<end clause> ::= end|<when clause>
```

2.9.1 Assignment Statements

The assignment statement assigns a new value to a variable thereby changing its value for subsequent references. For assignment to be properly defined:

1. The left-hand side must evaluate to a variable reference,

2. The mode of both sides must be identical, and

3. The value of the right-hand side must satisfy the

restrictions associated with the left-hand side variable reference. Whenever restrictions effect the size of the storage allocated, then the left- and right-hand restrictions must match exactly. For range restrictions it is only necessary that the right-hand value be within the limits of the left-hand restrictions.

4. The set of access restrictions placed on the type of the left-hand variable must be properly contained in the set of access restrictions for the right-hand expression.

Whenever it is not possible to check at parse-time that the restrictions will be be preserved by the assignment, then code will be generated to check the values during execution. If these run-time checks produce violations of the restrictions, then an appropriate error condition "assignerror" will be signaled.

<assignment statement> ::= <"type" variable reference> :=
<"type" expression>

Example:

```
Day := Thurs;
SyTab := Table(64 *: -1);
i := j // SyTab(j + SyTab(k))
M(i)(j) := k;
```

2.9.2 Routine Call Statements

A routine call statement serves to execute the body of the named routine after replacing all references to the formal parameters (defined in the procedure header) by the actual parameters (supplied with the procedure call). The correspondence is defined by matching the two lists pairwise in the order of appearance. A complete discussion of parameters is deferred to section 2.10.1.

<routine call statement> ::= <"routine name" id>
<actual parameter list>

Example:

```
p(x, y, z);
```

2.9.3 Leave Statements

The leave statement is used in connection with the loop statement to construct various loop termination tests. It states that the innermost loop statement is to be exited.

<leave statement> ::= leave

Example:

```
leave;
```

2.9.4 If-Then-Else Statements

The if-then-else statement has the traditional meaning: evaluate the boolean expression and perform the then clause if true and the else clause if false. By the addition of the end symbol the dangling else problem is avoided and the then and else clauses may use statement lists instead of single statements.

<if statement> ::= if <"boolean" expression> then
<statement list> {else <statement list>}! <end clause>

Example:

```
if
    day = Sun;
then
    weeks := weeks + 1;
    day := Mon;
else
    day := succ(day);
end;
```

2.9.5 Case Statements

The case statement provides an alternative to repeated if-then-else statements when there are several alternatives. The expression is evaluated to yield a value which then specifies at most one case clause to be executed. An optional else clause specifies what action is to be performed if none of the listed alternatives is met. If the else clause is omitted then one of the alternatives must always be met or a run-time error will result. The case labels must all be values of the same mode as the case expression and they must all be evaluatable at parse-time.

<case statement> ::= case <"simple" expression>
{<"simple" is clause>}+ {<else clause>}! <end clause>

```
<"simple" is clause> ::= is <"simple" value>
    {, <"simple" value>}* : <statement list>
<else clause> ::= else : <statement list>
```

Example:

```
case tax_bracket
is under_20k: taxes:= (20 * income) // 100;
is between_20k_30k: taxes:= (25 * income) // 100;
is between_30k_50k: taxes:= (30 * income) // 100;
else: taxes:= income;
end;
```

2.9.6 Loop Statements

The loop statement provides for all kinds of looping. By combining a loop statement with a leave statement (section 2.9.3) in various ways the usual "while", "until", "repeat", etc. statements can be constructed. Besides the increased flexibility of being able to place the loop test anywhere desired, separating the loop construct from its terminating condition creates a natural position for inserting a loop invariant immediately ahead of the termination test. Loop invariants will be discussed in section 3.6.2. After processes have been introduced is section 3.2.5, it will also be useful to construct loops without termination tests.

```
<loop statement> ::= loop <statement list> <end clause>
```

Example:

```
loop                                    {infinite loop}
    receive m from x;
    send m to y;
end;

loop                                    {until b do ... }
    if b then leave end;
    b := f(x);
    x := x + 1;
end;

loop                                    {repeat ... until b}
    b := f(x);
    x := x + 1;
    if b then leave end;
end;

i := m - 1; {quicksort A(m..n)} {n loops and a half}
```

```
j := n;
v := A(n);
loop
    loop
        i := i + 1;
        if not A(i) < v then leave end;
    end;
    loop
        j := j - 1;
        if not A(j) > v then leave end;
    end;
    if i > = j then leave end;
    A(i) := A(j);
end;
A(i) := A(n);
```

2.9.7 Begin Statements

In Gypsy the begin statement is unnecessary as a statement bracket because each of the other compound statements has been designed to allow a statement list instead of a single statement. However, there are one significant role for the begin statement in Gypsy: it is useful in the handling of exception conditions (section 3.7).

<begin statement> ::= begin <statement list> <end clause>

2.10 Routines

For the remainder of the current chapter, a routine is either a procedure, a function, or a program. In section 3.2.5, a fourth type of routine, a process, will be introduced. A routine is invoked by a routine call statement (2.9.2) and causes a private copy of the code for the routine to be executed with the formal parameters linked to the supplied actual parameters. In any actual implementation the actual code for the routine may be reentrant, but the data must be allocated uniquely with each invocation. This will become more important in section 3.2.5 after processes are defined.

Syntactically, the various routines differ only in their headers, but semantically there are significant differences. Each type of routine will be discussed individually in succeeding sections after a general discussion of parameters.

<routine unit> ::= <routine header> = <routine body>|

```
    <routine header> = pending
<routine header> ::= <procedure header>|<function header>|
    <process header>|<program header>
<routine body> ::= begin {<external spec> ;}!
    {<internal spec> ;}! {<local declaration> ;}*
    {<local spec> ;}! <statement list> <end clause>
<local declaration> ::= <local variable>|<local condition>|
    <local constant>|<local macro>|<local clock>
```

2.10.1 Parameters

There are three parameter passing limitations in Gypsy. All parameters are passed by reference from the calling environment, but depending upon the passing limitations specified in the formal parameter list parameters are treated differently in the routine environment.

Parameters which are designated as "var" parameters are conventional call-by-reference parameters. They may be both referenced and assigned and any assignment alters their value both locally and in the calling environment.

Parameters which are designated "const" parameters are call-by-reference parameters for which no alteration will be allowed in the routine environment. In short, they may be referenced but not assigned. This corresponds to the treatment of constants.

Parameters which are designated "copy" parameters are the conventional call-by-value parameters. In the environment of the formal parameter a copy will be made of the actual parameter upon routine entry. All subsequent references to the formal parameter are references to the copy of the actual parameter, not the actual parameter. The copy may be both referenced and assigned, but alterations will not effect the value of the actual parameter.

Whenever, the parameter limitations are not specified, the limitation "const" is assumed.

The allowable parameters in any routine call are variable references, non-variable-reference expressions, and conditions (not discussed until section 3.7). Whenever a non-variable reference is passed as an actual parameter the compiler will create storage for a value of the corresponding type and generate code to evaluate the non-variable reference storing the value into the created location. A reference to that location is then passed by reference. Since it would be impossible to recover any

result value placed at that location, the corresponding formal parameter in any call must be limited to "const" or "copy".

The address spaces of "var" parameters are never allowed to overlap other "var" or "const" parameters. In practice enforcement of this restriction will sometimes require run-time checks.

If a formal parameter type includes restrictions then these restrictions must be matched identically by the restrictions on the actual parameter type. However, if the formal parameter type is unrestricted then the formal parameter inherits the restrictions on the actual parameter type, which may vary from call to call. There are two ways of indicating that the formal type is unrestricted and that the actual restrictions are to be inherited: by the omission of the restriction and by the presence of a parameter declaration in place of the restriction. In both cases, the formal parameter inherits the actual parameter's restrictions; however, in the latter case the values of the restrictions are given names which may be used in the body of the routine. If an inherited restriction is subsequently used in a variable declaration which involves run-time allocation, then the restricted variable must meet any and all implementation restraints imposed on storage allocation. It will be necessary to perform this check at run-time.

If a formal parameter type includes an access list, then the access list for the formal parameter must be properly contained in the access list of the corresponding actual parameter (for an explanation of access lists see section 2.11.2).

Names introducted within a formal parameter list either as a routine parameter name or as a type parameter name are bound for the entire routine and hence, must be unique from other names over the scope of the routine.

```
<formal parameter> ::= {<limitation>}!
    <"type variable name" id> {,
    <"type variable name" id>}* :
    <"type" formal type reference>|
    cond <"condition name" id>  {, <"condition name" id>}*
<limitation> ::= var|const|copy
<formal parameter list> ::= ( <formal parameter>
    {; <formal parameter>}* )
<actual/formal type parameter list> ::=
    ( <actual/formal type parameter>
    {, <actual/formal type parameter>}* )
<actual/formal type parameter> ::=
    <actual type parameter>|<"type variable name" id> :
```

Example:

```
var x:integer
const m:Matrix
copy c:Color
```

2.10.2 Procedure Routines

Procedures form the basic unit for modular decomposition of programs.

```
<procedure header> ::= procedure {<access list>}!
    <"procedure name" id> <formal parameter list>
```

Example:

```
procedure exchange(var x, y:int) =
    begin
        var i:int;
        i:=x;
        x:=y;
        y:=i;
    end;
```

2.10.3 Function Routines

A function differs from a procedure in that it is referenced in the context of an expression and it returns a value. In addition, functions are not allowed "var" parameters. This restriction combined with the fact that there are no global variables insures that functions do not produce side effects. The lack of side effects not only aids verification, but eliminates the necessity to define the order of expression evaluation beyond that defined by the precedence relations. This improves the potential for code optimization.

Functions are allowed to return values of any type. The result of a function is the current value of the reserved variable "result" which is automatically declared to be of type matching the type of the function.

```
<function header> ::= function {<access list>}!
    <"type function name" id> {<formal parameter list>}!
    : <"type" type reference>
```

Example:

```
function Square(x:int):int =
    begin
        result := x ** 2;
    end;
```

2.10.4 Program Routines

There must be one and only one program routine in every assembled Gypsy program. Execution begins at the first executable statement in this program routine. A program routine can have only "var" parameters and only "var" parameters of type buffer or structures of type buffer (buffers will not be introduced until section 3.2.1). A program is treated as a special procedure which is called only once and which when exited causes the entire program to be terminated.

<program header> ::= program {<access list>}!
<"program name" id> <formal parameter list>

2.11 Gypsy Program

A Gypsy program is a series of units, each separately compiled and then collectively assembled into a complete program. A unit is a type unit, a constant unit, a routine unit, or a macro unit (section 3.3). Units may be entered in any order which allows for either top-down or bottom-up structured program development and facilitates readability by allowing a logical rather than functional ordering of the program.

Gypsy is designed to facilitate, but is not restricted to, interactive incremental program development. Units are compiled in the following fashion:

1. All currently defined unit names are entered into a data base along with their accompanying declarations. Each unit name must be unique.

2. A particular routine is selected for compilation (or verification).

3. As the compilation proceeds, names from parameters and local declarations are entered into the routine environment. In addition, whenever a declaration involving a type produces an unresolved reference, then the global data base is searched looking only at unit names. If the reference can be resolved,

then the rest of the declaration is retrieved from the data base and incorporated into the current global environment. Once a given unit has been retrieved, subsequent references will be satisfied locally without additional references to the data base. If a reference can be resolved by more than one unit definition, then the ambiguity produces a compilation error. Note that scalar type value names cannot be referenced directly without the type name being first referenced.

4. Any references not resolvable by accessing the data base as prescribed above are undefined references and are reported as errors.

5. Once the compilation is complete, the environment is erased before the compilation of the next routine. Any code generated is retained in the data base for later assembly into a complete Gypsy program. In an advanced implementation of Gypsy much of the environment information will also be retained in the data base to facilitate queries concerning unit interrelationships.

The advantages to the above compilation procedure are: lack of a required physical ordering of units, separate one-pass compilation of a given unit, and globally available routine and type names.

```
<Gypsy program> ::= {<unit> ;}+
<unit> ::= <type unit>|<routine unit>|
     <const unit>|<macro unit>
```

2.11.1 Scope

The scope of a name is related to the kind of entity that it names. In this section the scope of each different nameable entity will be discussed.

All type declarations are units; hence they are potentially globally defined. In the environment of any routine, type names are not automatically defined, but they are available for definition if they are referenced and there is no local declaration for the name. Whenever, and not before, there is a reference to the type, then the type name will become defined in the local environment along with all subcomponent names associated with the type, unless access is explicitly restricted (see section 2.11.2).

Constant, routine, and macro units are treated analogously. They are potentially globally available, but they are not defined

in the local environment until they are referenced. In all cases, references will be resolved locally whenever possible before looking globally. This allows for local renaming of unit names. Routines, etc. may also have restricted access (see section 2.11.2).

Once a unit name is referenced and subsequently becomes defined within a unit environment, then it has scope over the entire unit.

Parameter names have scope over the routine in whose header they appear.

Local declarations have scope over the begin statement in which they are declared.

Subcomponent names and scalar value names have scope equal to that of their type names, except as controlled by access restrictions discussed in section 2.11.2.

Given the above scope rules, the uniqueness rule for names is as follows: within any scope all base names (section 2.5.5) must be unique (i.e. all names must be unique except for modifier names).

2.11.2 Access Protection

An access list serves to limit access by explicitly stating the allowed accesses. By using an access list to restrict access, it is possible to provide all the protection customarily provided by a hierarchical language structure as well as the protection provided by "abstract" data structures plus more generalizable control without any of the accompanying unpleasantries. In this section, only the very rudimentary access protection concepts will be discussed.

An access list may accompany any unit declaration. It may appear in either of two positions within the unit header. First it may appear ahead of the unit name. In this position it names exhaustively those routines allowed to reference the unit name. For routines, it lists those routines which are allowed to call the routine; for types, it lists those units which are allowed to declare variables of that type; etc.

The second position in which an access list may occur is immediately preceeding the "=" symbol in the unit header. In this position, it restricts access to the information available in the unit body. For other than type units this offers no

apparent significance and hence is not allowed, but for types it allows the declaration of variables of a specified type without being allowed knowledge of the internal structure of the object. The effect is the construction of an "abstract" type.

An abstract type consists of an "abstract" object with a set of operations which manipulate the "abstract" object. The abstract object is actually implemented via other types, but only the operations defined on the abstract object are allowed to know the particulars of the implementation. Hence, the abstract type appears to be an indivisible "primitive" type in the language.

Access lists are always optional and when omitted it is assumed that every unit has complete access capabilities.

In compiling a routine, when it is necessary to search the data base for a unit name the effect of the access lists is as follows:

1. If the access list appears ahead of the unit name and the routine being compiled is not included in the access list, then the name as well as its declaration are hidden from the routine and will appear undefined.

2. If the access list appears on the end of the header and the routine being compiled is not included in the access list, then the information available in the header is available to the routine, but information internal to the declaration remains hidden. Hence, subcomponent names do not get defined and references to them will produce errors. Likewise, references through functions, operators, or indices require knowledge of the internal structure which is unavailable; hence, they generate errors.

```
<access list> ::= <"unit name" id>
    {, <"unit name" id>}*
```

Example:

```
type intstack <push,pop,top> =
    record(st:array((1..10)) of int;
           pt:int);

procedure push(var s:intstack; e:int;
        var overflow:boolean) =
    begin
        overflow := false;
        s.pt := s.pt + 1;
        if s.pt <= 10
        then s.st(s.pt) := e
        else overflow := true
```

```
        end
    end;
```

By defining combinations of types employing identical modes and restrictions, but with different access lists, it is possible to selectively limit a routines access capabilities. For example, suppose routine A declares a stack. It then has available the routines push, pop, and top for manipulation of the stack. Further suppose that routine B, which is called by A, should only reference routine top and that we would like to restrict B is such a way that the compiler can assure us that this is indeed the case. This is accomplished by the following example.

Example:

```
procedure A(...) =
    begin
        var s: intstack;
        ...
        B(s);
        ...
    end;

type rintstack<top> = intstack;

procedure B(x: rintstack) =
    begin
        ...
        {only top(x) legal here}
        ...
    end;
```

The semantics of actual/formal parameter pairs with different access lists is simply that the access list associated with the actual parameter type must contain the access list associated with the formal parameter type.

2.12 Further Examples

The following examples are a collection of routines which define and manipulate a hash table. The example was borrowed from the Euclid Report [14]. The example solution presented here uses a few features which have yet to be discussed, but it is believed that these features are generally understandable.

```
type CyclicScan<init,next,value,stop> =
```

```
    record
        (initial: boolean;
         value,start,limit: bounds);

function <search,delete,insert> init(item: int;
        size: bounds): CyclicScan =
    begin
        result.initial:=true;
        result.value:=hash(item,size);
        result.start:=result.value;
        result.limit:=size;
    end;

procedure <search,delete,insert> next(var i:CyclicScan) =
    begin
        if i.initial then i.initial:=false
        else
            if i.value=i.limit
            then i.value:=1
            else i.value:=i.value+1
            end;
        end;
    end;

function <search, delete, insert>
        val(i: CyclicScan): bounds =
    begin
        result := i.value;
    end;

function <search, delete, insert>
        stop(i: CyclicScan): boolean =
    begin
        result := (not i.initial) and
            (i.value = i.start)
    end;

const large: integer = pending;

type bounds = integer (1..large);

function hash(item: int;
        size: bounds): bounds = pending;

type HashTable(size: bounds)
        <initHash,search,delete,insert> =
    array(integer(1..size)) of record
        (flag: (fresh,full,deleted);
         key: int);

function initHash(table: HashTable(size: bounds))
```

```
        : HashTable(size) =
    begin
        result := HashTable(size *: _(fresh, 0))
    end;

function search(key: int;
        table: HashTable(size:bounds)): boolean =
    begin
        var i: CyclicScan:=init(key,size);
        result:=false;
        loop
            next(i);
            if stop(i) then leave end;
            case table(val(i)).flag
                is fresh: leave;
                is full: if table(val(i)).key=key
                            then result:=true;
                         leave;
                     end;
                is deleted: ;
            end;
        end;
    end;

procedure delete(key: int;
        var table: HashTable(size:bounds)) =
    begin
        var i: CyclicScan:=init(key,size);
        loop
            next(i);
            if stop(i) then leave end;
            case table(val(i)).flag
                is full: if table(val(i)).key=key
                    then table(val(i)).flag:=deleted;
                         leave;
                     end;
                is fresh: leave;
                is deleted: ;
            end;
        end;
             end;

procedure insert(key: int;
        var table: HashTable(size:bounds)) =
    begin
        var i: CyclicScan:=init(key,size);
        if not search(key,table) then
            loop
                next(i);
                if stop(i) then leave end;
                case table(val(i)).flag
```

```
                    /is fresh,deleted:
                         table(val(i)).flag:=full;
                         table(val(i)).key:=key;
                         leave;
                     is full: ;
                end;
            end;
            if stop(i) then error("table full") end;
        end;
    end;
```

Chapter 3

VERIFIABLE SYSTEMS LANGUAGE

In Chapter 2, the basic language without extensions for concurrency, verification, or error handling was defined. In this chapter of the report these extensions will be developed. Together, the combination of Chapter 2 and the present chapter are considered the minimal definition of Gypsy.

The features included in this section stem primarily from two sources: those developed through operating systems applications to support concurrency and those developed to support verification of programs written in Gypsy. No feature has been included in Gypsy for which there was not also included some facility to support verification of its properties.

3.1 Extended Types

In addition to the "conventional" types defined in Chapter 2, Gypsy provides an extensive set of "list" types. These coupled with facilities for "abstract" and "parameterized" types provide the necessary support for program verification as well as structured programming.

3.1.1 Type Parameters

Parameterized type declarations allow for construction of generic types which have analogous definitions. There are two categories of type parameters: modes and restrictions (recall that a type consists of both a mode and a set of restrictions). Only restriction parameters will be allowed.

Syntactically, type parameters are introduced in the type header almost identically to the way routine parameters are introduced. The most significant difference is that type parameters need no limitations; hence, no "var" etc. designations.

To illustrate the usage of type parameters, recall the **stack** declaration in section 2.11.2. Without the ability to **parameterize** the type, stack type was declared with a fixed sized

array. This definition can be made more flexible by leaving the array size unspecified, to be supplied as a restriction in a subsequent declaration. For example,

```
type intstack(n:integer)<push,pop,top> =
    record(
        st:array((1..n)) of integer;
        pt:(0..n));
```

might be used in the subsequent declarations:

```
var s:intstack(50);
var t:intstack(100);
```

Now observe the corresponding change in procedure push.

```
procedure push(var s:intstack(n:);
        e:integer; var overflow:boolean) =
    begin
        overflow := false;
        s.pt := s.pt + 1;
        if s.pt <= n
        then s.st(s.pt) := e
        else overflow := true
        end
    end;
```

Since restriction type parameters only effect restrictions and not modes, there is little problem in generating code for routines manipulating these parameterized types.

```
<formal type parameter list> ::= ( <formal type parameter>
    {; <formal type parameter>}* )
<formal type parameter> ::= <"simple variable name" id>
    {, <"simple variable name" id>}*
    : <"simple" formal type reference>
```

A reference to a parameterized type looks like a routine call, i.e. the type name followed by an actual parameter list. However, there are some differences. In certain instances parameters in type instantiations may be omitted. Whenever all the parameters are omitted, then the entire parameter list may be omitted. Thus,

```
var s : intstack();
var s : intstack;
```

both of the above are equivalent.

Parameters may be omitted under the following circumstances:

1. Formal parameter types need not be completely specified and thus may have missing restriction parameters. The missing parameter values will be extracted from the corresponding actual parameters.

2. Variables used within formal proof specifications may have missing parameters. In each case all values for the given mode will be implied. For integers this implies an infinite number of values.

3.1.2 List Types

The list types - sequences, strings, sets, and bags - were included of necessity for program specification, but they are generally useful types and are available for variable declarations as well.

A sequence is an ordered list to which objects can be added or removed at either end. A sequence is declared by specifying the type of the component elements. By definition a sequence can be arbitrarily long even if the component type is finite since any value may appear arbitrarily often within the sequence. For purposes of implementation it is highly desirable that sequences be bounded as it is then possible to statically allocate storage for the maximum size of the sequence. Hence, while arbitrary sequences are allowed within proof-time evaluations, only restricted sequences are allowed for run-time evaluation.

In Gypsy, sequences are bounded by placing a size restriction on the declaration. The effect is to bind the maximum number of elements allowed in the sequence.

```
<"sequence of type" type declaration> ::= sequence <size>
    of <"|ype" type declaration>
<size> ::= ( <"integer" expression> )|{empty}
```

Example:

```
type DaysOfYear = sequence  (365) of DaysOfWeek;
type BitString(n:integer) = sequence (n) of boolean;
type SixBitString = BitString(6);
```

A string is a special predefined sequence type.

```
type string(n:integer) = sequence (n) of character;
```

Because it is predefined, the compiler is aware of its special status and depending upon the implementation may implement strings differently from other sequences.

Bounds for strings are indicated in exactly the same manner as for sequence types, i.e. by following the string type name by a size restriction.

Example:

```
type Message = string(50);
```

A set is an unordered list in which all duplications have been collapsed. Sets of any finite type are necessarily finite; however, they may be extremely large. Hence, Gypsy allows size restrictions to be placed upon a set declaration to limit the amount of storage actually allocated.

```
<"set of type" type declaration> ::= set <size> of
    <"type" type declaration>
```

Example:

```
type SetOfDays = set of DaysOfWeek;
type StartingLineUp = set (5) of integer(1..99);
```

A bag (also called a multi-set) is an unordered list in which all duplications have been preserved. Hence, a bag is like a sequence in that even when the component type is finite the bag is potentially unbounded. Thus bags declared for run-time evaluation are required to be restricted.

```
<"bag of type" type declaration> ::= bag <size> of
    <"type" type declaration>
```

Example:

```
type BagOfSetOfDays = bag (100) of SetOfDays;
type BagOfStrings = bag (10) of string(100);
```

3.1.3 List Values

A value composition for any of the list types is a possibly empty list of values of the component type. To compose a value for one of the list types, the type name is used as a function name with a parenthesized list of values of the correct component type. This is analogous to the manner in which values for

records and arrays are composed, except that the number of component values is flexible. If their are no values within the parentheses, it is interpreted as an empty list.

```
<"list of type" composition> ::=
    {<"list of type type name" id>}! ( {<"type" component>
    {, <"type" component>}*}! )
```

Example:

```
Message()
StartingLineUp(11,19,22,14,33)
```

String values may also be expressed as strings of characters within quotes.

```
<"string" value> ::= <string>
```

Example:

```
"This is a string value."
"So is this:" "x"
```

3.1.4 List Operations

The operators defined here are extensions of previous operators, new operators extending previous operator classes, and new operators in new operator classes. The new operator classes sequence, set and bag all have precedence equal the class of integer operators. Within each of the new classes all operators have equal precedence.

Of the operators defined in section 2.8.2 only equality and inequality apply to list types as well. Two sequences are equal if and only if they are of the same length and they contain the same values in the same order; two sets are equal if and only if they contain the same values; and two bags are equal if and only if they contain the same values and for each value they contain the same number of copies of that value.

```
<"boolean" expression> ::= <"list" expression> eq\=
    <"list" expression>|<"list" expression> ne\<>
    <"list" expression>
```

Example:

```
"abc" = "abc"
"abc" <> "a b c"
```

There are two relational operations which apply to all list types. These are the sub and member relations. The sub relation evaluates to true if one list value is a sublist of another. The member relation evaluates to true if a component value is a member of a specified list value composed of components of that component type.

```
<"boolean" expression> ::= <"list" expression>
    sub <"list" expression>|<"type" expression>
    member <"list of type" expression>
```

Example:

```
"def" sub "abcdefghij"
Mon member SetOfDays(Mon, Wed, Fri)
```

There is one additional sequence operator, called append, which concatenates two sequences of the same mode.

```
<"sequence" expression> ::= <"sequence" expression>
    append\@ <"sequence" expression>
```

Example:

```
"this" @ "is" @ "a" @ "string." = "thisisastring."
```

There are three additional operations which apply to sets and bags. These are union, intersection, and difference.

```
<"set\bag" expression> ::= <"set\bag" expression>
    union <"set\bag" expression>|<"set\bag" expression>
    intersect <"set\bag" expression>|<"set\bag" expression>
    difference <"set\bag" expression>
```

Example:

```
SetOfDays(Tues) union SetOfDays(Thurs)
```

3.1.5 List Functions

Portions of sequences are referenced through four functions: first, last, nonfirst, and nonlast. First returns the value which is the first value of the given sequence (conceptually the left hand end of the sequence) and last returns the value which is the last value of the sequence (i.e. the right end). Nonfirst and nonlast are the complementary functions to first and last, respectively. Namely, the nonfirst of a sequence is the sequence minus its first value and the nonlast is the

sequence minus its last value. Thus if "s:t" is a sequence declaration, then these functions satisfy the following relationships:

```
t(first(s)) @ nonfirst(s) = nonlast(s) @ t(last(s)) = s
```

There is one additional sequence function length which returns an integer value equivalent to the number of values in the sequence.

```
<"type" expression> ::= first (
    <"sequence of type" expression> )|last (
    <"sequence of type" expression> )
<"sequence" expression> ::= nonfirst (
    <"sequence" expression> )|nonlast (
    <"sequence" expression> )
<"integer" expression> ::= length (
    <"sequence" expression> )
```

Example:

```
seq1 = seq2 iff length(seq1) = length(seq2) and
    seq1 sub seq2 and seq2 sub seq1
```

Gypsy includes one rather unusual function, called ismerge, which determines if there exists a merge of a bag of sequences which is equal to a given sequence. This function is extremely valuable in writing specifications for concurrent processes. Because of its existential nature, this function is only allowed in specifications which will not be executed.

```
<"boolean" expression> ::= ismerge (
    <"sequence" expression> ,
    <"bag of sequence" expression> )
```

Example:

```
ismerge("abcdefgh", BagOfStrings("bcf","e","ad","gh"))
```

3.2 Concurrent Processes

Gypsy was specifically designed to support systems software, particularly communications software, which demands facilities for describing concurrent execution. To be consistent with Gypsy's philosophy of allowing nothing in the language which formal verification could not handle and of providing an extensive enough programming language that would eliminate the need to go outside of the language required a powerful, yet

tightly controlled, set of language features. These features are described in the next sections.

3.2.1 Buffer Types

A buffer is a special type of list type which is used only for interprocess communication. It is the only type of parameter which may be shared between processes. Furthermore it is the only type of variable which cannot be passed as a "copy" parameter.

Associated with every buffer variable is a semaphore which guarantees exclusive access of a single process to the entire variable during the active execution of any operation on the buffer variable. There is also a pair of FIFO queues associated with each buffer. The first is a queue of processes waiting for exclusive access and the second is a queue of processes waiting for some property of the buffer to change (depending upon the operation).

Buffers may also be used to communicate with the external environment; a fact which is exploited for input/output (see section 3.5).

It is important to note that since Gypsy programs may be distributed across several machines, communicating processes may or may not reside on the same machine. In either case the compiler must provide the necessary support for the buffer operations.

While the syntax appears to allow arbitrary structures intermixing buffers with other types, the compiler will only allow "pure" buffer structures, i.e. if any component of a structure is a buffer then all components of the structure must be buffers. This prevents problems that would arise in passing shared buffer structures.

```
<"buffer of type" type declaration> ::= buffer <size> of
    <"type" type declaration>
```

Example:

```
type MessBuf(n:integer) = buffer (n) of Message;
var Buf:MessBuf(1);
var m:Message;
```

3.2.2 Buffer Procedures

Because of the unusual nature of buffers, several operations usually available for variables are not allowed. There is no way to explicitly initialize a buffer; however, by definition all buffers are initially empty and their FIFO queues are empty. There are no comparison operations; even equality is not defined. Assignment to a buffer variable is not allowed. The only allowable operations are described below.

Receive and send are the two primitive operations of process communication. The receive operation

```
receive m from Buf;
```

returns in m the first entry of the queue associated with Buf provided Buf is nonempty. However, if Buf is empty the calling process is indefinitely suspended and placed in a FIFO queue pending further entries into Buf. When the process reaches the first entry of the FIFO queue and an entry is made in Buf, then the process is reactivated; it dequeues the first entry in Buf and returns. Similarly,

```
send m to Buf;
```

appends m on the end (last) of the buffer list associated with Buf provided Buf is not full. If Buf is full then the calling process is suspended and placed in a FIFO queue pending removals from Buf. When the process becomes the first entry of the FIFO queue and Buf is reduced below capacity, then the process is reactivated; it enqueues m and returns.

```
<buffer statement> ::= send <"type" expression> to
    <"buffer of type" variable reference>|receive
    <"type" variable reference> from
    <"buffer of type" variable reference>
```

At any time that a process is suspended either on a receive or a send the process is said to be "blocked". The specification of concurrent processes (section 3.6.3) is based upon statements about conditions during blocked intervals.

3.2.3 Buffer Histories

Each buffer has associated with it a number of histories which record the results of the send and receive operations performed on the buffer. For a buffer that is declared as

```
var b:buffer(bsize) of btype
```

the mode of each history associated with b is

```
sequence of btype.
```

Histories provide a precise way of stating formal specifications for concurrent processes, and it is not necessary that they be implemented.

Each buffer is viewed as a record consisting of the following components

```
record(allto:sequence of btype;
       bufq:sequence(bsize) of btype;
       allfrom:sequence of btype)
```

The allto field is the sequence of all objects sent to the buffer, bufq is the content of the buffer queue, and allfrom is the sequence of all objects received from the buffer. Except during send and receive operations, these components satisfy the axiom

```
b.allto = b.bufq @ b.allfrom.
```

Upon initial creation of a buffer, all three components are the empty sequence.

The three buffer components given above are referred to as "global" histories because they record all transactions, by any process, on the buffers. For specifications, it also is necessary to have "local" histories that record just the transactions of a particular process p on a buffer. There are two of these histories that can be used, b.infrom and b.outto. B.infrom is like b.allfrom except that it records only those objects that are received from b by process p. B.infrom, therefore, is always a subsequence of b.allfrom. B.outto has a similar relationship to b.allto. Upon entry to process p, both b.infrom and b.outto are empty. They exist during the execution of process p, and do not exist upon exit from p. Examples of these histories appear in the switching network in Chapter 4.

3.2.4 Buffer Functions

There are two boolean functions empty and full which test the status of a buffer, but since they are only meaningful when a process is blocked; they can only be used in proof-time blockage specifications.

```
<"boolean" expression> ::= empty (
    <"buffer" variable reference> )|full (
    <"buffer" variable reference> )
```

3.2.5 Process Routines

Processes differ from procedures only in the manner of their invocation and in the limitations on their parameters. Concurrent processes are invoked by a cobegin statement and are executed asychronously with all other processes listed within the cobegin. For instance,

```
cobegin
    p(...);
    q(...);
    r(...);
end;
```

stipulates that the three processes p, q, and r are to be executed concurrently. The calling routine is suspended as a result of the process calls until either all the processes exit and return or an exception condition occurs.

Process scheduling is performed by the run-time support system and is incompletely specified here. The chosen algorithm must satisfy at least these three criteria: 1. the system must not block as long as there are still processes capable of being run, 2. the algorithm must support priority levels, and 3. the algorithm must be "fair" among equal priorities. The first stipulation is required for verification; the second stipulation provides a solution to interrupts; and the last stipulation is highly desirable of any scheduling algorithm.

Interrupts are handled in Gypsy by queuing a message to a high priority process suspended waiting for input from the interrupting device. The scheduling algorithm must respond by preempting any lower priority process and immediately scheduling the interrupt process.

Note: More information on the processing of interrupts will be supplied in a supporting document on compiler directives [11].

Since there are no global variables, the only variables that processes can share are parameters; and, since buffers are the only types of variables which are protected by semaphores, they are the only alterable parameters allowed to processes, i.e. the only "var" parameters. This insures that all shared

variables, except constants, are protected. Other types of variables may be passed as "const" or "copy" parameters to processes.

Processes can invoke functions, procedures, and other processes; however, processes may not be recursive. When a process invokes another routine, it invokes a separate copy which is not shared with any other processes. In terms of the run-time environment, process invocation causes a fork, creating a new environment for each concurrent process. Each process then executes other routines by envoking them in their own process environment. In any actual implementation reentrant code is to be encouraged and will limit the number of required copies to separate copies of the data plus one copy of each routine per machine.

Note: More information on the distribution of code across machines will be supplied in a supporting document on compiler directives [11].

<process header> ::= process {<access list>}!
<"process name" id> <formal parameter list>

Example:

```
process producer(var get, store: Buf) =
    begin
        var i: Message;
        loop
            receive i from get;
            send i to store;
        end;
    end;

process consumer(var store, put: Buf) =
    begin
        var i: Message;
        loop
            receive i from store;
            send i to put;
        end;
    end;

program ProducerConsumer(var get, put: Buf) =
    begin
        var store: Buf;
        cobegin
            producer(get, store);
            consumer(store, put);
        end;
    end;
```

3.2.6 Cobegin Statements

The cobegin statement is used solely to start concurrent execution of processes. It stipulates that each of the process calls contained within it are to be invoked simultaneously. From an implementation point of view, it stipulates that each process is to be scheduled for execution and that the calling routine is to be suspended pending termination of the called processes.

```
<cobegin statement> ::= cobegin
    <extended routine call statement>
    {; <extended routine call statement>}* <end clause>
```

Example:

```
cobegin
    Producer(get, store);
    Consumer(store, put);
end;
```

3.2.7 Extended Routine Calls

Because it may be desirable to call some number of identical copies of a process, it is possible to parameterize a process call and stipulate a range of values to be substituted for the parameter. This construct is only valid within a cobegin statement. The local index has scope only over the process call statement.

```
<extended routine call statement> ::=
    <routine call statement> {<each clause>}!
<each clause> ::= each <"type variable name" id>
    : <"type" type reference>
```

Example:

```
cobegin
    Producer(get(i), store(i)) each i:(1..N);
    Consumer(store(i), put(i)) each i:(1..N);
end;
```

3.2.8 Await Statements

The await statement creates a facility for simultaneously waiting on more than one buffer operation. This is an essential

capability for any polling process. The optional after clause allows the waiting process to resume execution after a specified time interval even if none of the buffer operations has completed. The first buffer operation capable of completing will be performed followed by the corresponding on clause. The algorithm for selecting between two or more operations capable of being performed is assumed to be "fair" so that considerations of indefinite waiting may be ignored.

```
<await statement> ::= await {on <buffer statement> :
    <statement list>}+ {after <"integer" expression> :
    <statement list>}! <end clause>
```

Example:

```
loop
    await
    on receive m from x : send m to y;
    on receive m from y : send m to x;
    after longtime: send "start" to x;
    end;
end;
```

3.2.9 Clock Types

A clock variable is a data structure realization of time. It provides a means of measuring time and of synchronizing events. In Gypsy, clocks are special variables which the programmer is never allowed to alter, but which monotonically increase periodically.

Since Gypsy programs may be distributed across a network of computers, where it would be impossible to synchronize clocks, and since Gypsy programs are written without regard to how they will be distributed, there are several stipulations on the usage of clocks.

1. There are no global clocks (consistent with no global variables).

2. Clocks cannot be passed as parameters (caller and callee may not be on the same machine).

3. Two different clock variables are not guaranteed to be synchronized (in fact, they may not even increment at the same rate).

Clock variables are referenced by a special clock statement

which results in an integer value being assigned to an integer variable. A clock statement was chosen in lieu of a function in order to avoid expression order evaluation problems as well as comparison expressions, such as clock = clock, which are sensitive to timing.

There is no facility for initializing clocks. Clocks run independent of any program or programmer control.

Clock declarations can occur as local declarations in processes, procedures, and programs, but not in functions.

<local clock> ::= clock <"clock variable name" id>
{, <"clock variable name" id>}*

Example:

clock time;

3.2.10 Clock Statements

A clock statement is used to obtain the current time as recorded by a named clock variable.

<clock statement> ::= log <"integer" variable reference>
at <"clock variable name" id>

Example:

log i at time;

3.3 Macros

A macro provides a means of assigning a name to a parameterized expression. It is parameterized, but there are no type declarations for the parameters. Macros are compiled by replacing the calling reference by the macro body after substituting the actual parameters for the formal parameters, and then evaluating the expression in the calling environment. Thus any free variables in the macro body are bound in the calling environment.

A macro may appear in any context in which a variable reference or expression is legal, including on the left of an assignment provided that the macro once replaced evaluates to a variable reference.

Macros, like constants, can appear as either a unit or as a local declaration.

Macro calls may be nested, but recursion is forbidden.

```
<macro unit> ::= <macro header> = <"untyped" expression>
<local macro> ::= define <"macro name" id>
    {( <"untyped variable name" id>
    {, <"untyped variable name" id>}* )}!
    = <"untyped" expression>
<macro header> ::= define {<access list>}!
    <"macro name" id> {( <"untyped variable name" id>
    {, <"untyped variable name" id>}* )}!
```

Example:

```
begin
    define Taxes(i) = TaxFile(i).Tax;
    i := 1;
    loop
        if i > N then leave end;
        TotalTaxes := TotalTaxes + Taxes(i);
        i := i + 1;
    end;
end;
```

3.4 Pending

The reader is reminded that Gypsy was designed for structured, interactive, and incremental program development and verification. A valuable development aid is the "pending" expression/statement. It allows a partial development of a unit for which the programmer wishes to postpone complete specification and/or implementation. The intent is to allow partial information to be entered into the data base and employed without the compiler objecting to the missing parts. For verification it is frequently valuable to have the unit header and certain specifications available even without the actual code. The language is designed to allow partial compilation of units even in the presence of pendings; however, there will be no attempt at code generation and the data base will be marked to record the presence to the pendings for later prompting.

Example:

```
type Surprise : SurpriseType = pending;

function Pop(s:intstack(n:)):integer =
```

```
begin
    cexit Pop(s) = s.st(s.pt);
    pending;
end;
```

3.5 Input/Output

Input/output in Gypsy is accomplished by process communication with external processes. The buffers used to perform this external communication are parameters to the program routine. These buffers must be declared to match the data format of the desired devices.

Note: More information on input/output will be supplied in the supporting document on compiler directives [11].

3.6 Specifications

In this report no attempt will be made to develop the methodology for verifying programs, only the necessary specification statements will be defined. The proof methodology will be described in detail in other supporting documents.

Specifications fall into two categories: those that are to be proved and those that are to be validated. Syntactically, these two categories are distinguished by the absence or presence of an "otherwise" clause. Whenever the optional clause is present the specification is to be checked at run-time and if it should fail to be true, then the named condition is signaled.

More specifically, there are four possibilities for specifications. These are 1. prove, 2. prove and check, 3. assume, and 4. assume and check. In addition there are two shorthand variations which allow the keywords "prove" and "assume" to be omitted. These two possibilities are defined by: "... b" is short for "... (prove b)" and "... b otherwise c" is short for "... (assume b otherwise c)", where "..." is one of the specification keywords.

Whenever a primed variable reference appears within a specification statement it is interpreted as the value of the variable reference upon entry to the routine and are allowed only in "keep", "exit", "cexit", and "block" specifications. Primed variable references are not allowed in run-time expression evalutions.

```
<assertion> ::= <"boolean" expression>|
    <"boolean" expression> otherwise <condition>|
    ( <assertion clause> {; <assertion clause>}* )
<assertion clause> ::= prove <"boolean" expression>|
    prove <"boolean" expression> otherwise <condition>|
    assume <"boolean" expression>|
    assume <"boolean" expression> otherwise <condition>
<"type" primed variable reference> ::=
    <"type" variable reference> '
```

3.6.1 Type Specifications

There are two kinds of type specifications: requirements and axioms. Requirement specifications are necessitated by the existence of type parameters. They allow the specification of required properties which must be met in order for the type definition to be properly defined. An axiom states properties about the data type which must hold at all times except when the structure is being manipulated by any of the routines explicitly named in the type access list.

```
<type spec> ::= {require <assertion>}!
    {axiom <assertion>}!
```

Example:

```
type intstack(n:integer) <push, pop, top, empty, equal> =
    begin
    require n > 0;
    axiom all s:intstack(n),
        not empty(s) imp equal(push(pop(s), top(s)), s);
    record(
        st:array(integer(1..n)) of integer;
        pt:integer(0..n));
    end;
```

3.6.2 Statement Specifications

Statement specifications make assertions about the state of the program at certain points in the code. These specifications may be checked by verification prior to execution, validated during execution, or analyzed after execution depending upon the type of statement. Assertions state properties which must hold whenever control passes through that statement in executing the program. The trace statement calls for a snapshot dump of the variables involved in the expression at the point of execution

for the statement. These dumps will be followed by a post-execution analysis.

```
<statement spec> ::= assert <assertion>
    trace <"boolean" expression>
```

Example:

```
procedure concat(var a, b: string) =
...
loop
    assert a @ b = a' @ b'
    if length(b) = 0 then leave end;
    trace length(b) > 0;
    a := a @ b.first;
    b := b.tail;
end;
```

3.6.3 Routine Specifications

Routine specifications all have the same basic form. They differ only in the scope of their definitions. We will comment about each individually.

"Entry" specifications are valid only upon routine entry; likewise, "exit" specifications are valid only upon routine exit. "Block" specifications are true only when a routine is suspended due to a buffer condition. In stating external specifications, only variables declared outside the block may be referenced. With respect to parameters, external specifications are only allowed to reference identifiers declared in the routine header; never internal names of their type declarations.

"Centry", "cblock", and "cexit" specifications (concrete entry, etc.) perform the same function as "entry", "block", and "exit" specifications except that in addition to the information available in the routine header they may also reference the type body definitions referred to in the header, provided of course that the routine has access rights (see section 2.11.2). The significance of this distinction is that if a routine x references a routine y in which there are both external and internal specifications, then the external specifications of y may be extracted and used in the proof of x, but not so for the internal specifications of y.

Both "exit" and "cexit" specifications are allowed a special case syntax to enumerate the possible exit conditions that correspond to normal as well as conditioned termination.

The reserved identifier "normal" is used to identify the normal case.

A "keep" specification is used to make statements about the properties of local variables which are true after variable initialization for the duration of their scopes. Unlike other routine specifications this specification is always valid within the routine in which it is declared. The keep does not, however, follow parameters into the execution environment of a called routine.

Routine specifications are evaluated in the following order: entry, centry, {..., cblock, block,} ..., cexit, exit; where "..." represents routine code and with keep specifications being evaluated where appropriate.

```
<external spec> ::= {<entry spec>}! {<block spec>}!
    {<exit spec>}!
<entry spec> ::= entry <assertion>
<block spec> ::= block <assertion>
<exit spec> ::= exit <assertion>|exit case
    ( {is <condition> {, <condition>}* : {<assertion>}!}+ )
<internal spec> ::= {<centry spec>}! {<cblock spec>}!
    {<cexit spec>}!
<centry spec> ::= centry <assertion>
<cblock spec> ::= cblock <assertion>
<cexit spec> ::= cexit <assertion>|cexit case
    ( {is <condition> {, <condition>}* : {<assertion>}!}+ )
<local spec> ::= keep <assertion>
```

Example:

```
var i:integer := 0;
keep i mod 2 = 0;     (i is an even number)
```

Example:

```
procedure push(var s:intstack(n:);
        const e:integer; cond overflow) =
    begin
        exit case(
            is normal: top(s) = e
            is overflow: s = s');
        cexit case(
            is normal: s.pt = s'.pt + 1 and s.st(s.pt) = e
            is overflow: s = s');
        if s.pt < n
        then s.pt := s.pt + 1;
            s.st(s.pt) := e
        else signal overflow
        end;
```

```
        end;
```

Example:

```
    program ProducerConsumer(var get, put:Buf) =
        begin
            block not full(put) imp empty(get) and
                put.outto = get.infrom;
            var store:Buf;
            cobegin
                producer(get, store);
                consumer(store, put);
            end;
        end;
```

3.7 Run-Time Validation

Run-time validation serves the function of verifying specifications by performing run-time checks to establish their validity. A specification is specified for run-time validation by appending an otherwise clause to the specification. The compiler will then generate the appropriate code to perform the checks and signal the specified condition should it fail. This condition must be declared in an enclosing scope.

3.7.1 Conditions

A condition is an instantaneous event which may occur during the execution of a program; caused by either a predefined or a user-defined event. The exact set of predefined conditions depends in part upon actual hardware design, but a subset of necessary conditions can be defined independent of hardware peculiarities. The list enclosed is by no means exhaustive. A condition declaration creates a name for a user-defined condition which the user may cause by explicitly signalling it or by its being attached to a run-time check. Conditions may be passed as parameters in routine calls.

```
<local condition> ::= cond <"condition name" id>
    {, <"condition name" id>}*
<condition> ::= <"condition name" id>|indexerror|caseerror|
    aritherror|initializeerror|buffererror|routineerror|
    rangeerror|senderror|receiveerror|assignerror|typeerror
```

Example:

```
cond stackoverflow;
```

3.7.2 Signal Statements

The signal statement provides the user a means of causing a condition. The user caused condition is treated exactly like a system caused condition.

<signal statement> ::= signal <condition>

Example:

```
signal stackoverflow;
```

3.7.3 Condition Clauses

All compound statements terminate with an end clause. Up to this point in this report, the end clause has been simply the word end. However, an end clause may also be an optional "when" clause. A "when" clause lists a set of conditions with corresponding actions to be taken in the event that the condition should occur. The "when" clause is treated exactly like a case statement where the case expression type is a condition and the case labels are condition names. For normal exits from a compound statement, the "when" clause is skipped over and execution continues immediately following the "end" of the compound statement.

Whenever a condition occurs inside a compound statement the following actions occur:

1. If the condition name appears in the "when" clause for the compound statement, then the corresponding action is executed.

2. If the "when" clause for the compound statement has an else part, then the else action is executed.

3. If the compound statement is contained within another compound statement, then the inner compound statement is exited to the next immediately enclosing compound statement and the search continued from step 1.

4. If there are no more enclosing compound statements and if the condition name is a parameter, then the routine is exited and the search is continued from step 1 in the calling

environment after substituting the actual for the formal condition parameter name.

5. If there are no more enclosing compound statements and the condition is not a parameter, then the routine is exited with the condition being transformed into a routineerror. The search continues from step 1 in the calling environment.

6. If there is no calling routine, i.e. it is the main program, then the system aborts.

A "when" clause is only entered upon the occurrence of a condition and is ignored under normal execution. Whenever an action within a "when" clause is executed, then the compound statement is exited immediately afterwards.

If the compound statement being exited happens to be a cobegin statement an interesting situation occurs. What happens to the other processes within the cobegin? If any process within a cobegin returns a condition, all of the other processes are immediately signalled by a cobeginerror condition to terminate. After all processes have terminated, then the original condition is processed in a normal manner beginning at step 1. At the moment there is no scheme for enforcing termination of a process voluntarily.

```
<when clause> ::= when {is <condition> {, <condition>}*
    : <statement list>}+ {else : <statement list>}! end
```

Example:

```
loop
    x := x + 1;
when
is rangeerror : x := 0;
end;
```

Chapter 4

A MESSAGE SWITCHING NETWORK

The following example follows part of the development of a simple message switching network and illustrates many of the important features of Gypsy. Only the specification and implementation of the network will be discussed. Its verification is beyond the current scope. The development of the network will be top-down, but Gypsy admits any kind of program design strategy.

The top-level structure of the network is shown in figure 1. The network switches messages among a fixed number of users, each of which communicates with the network through a port. We will ignore protocols, and assume that each message is a

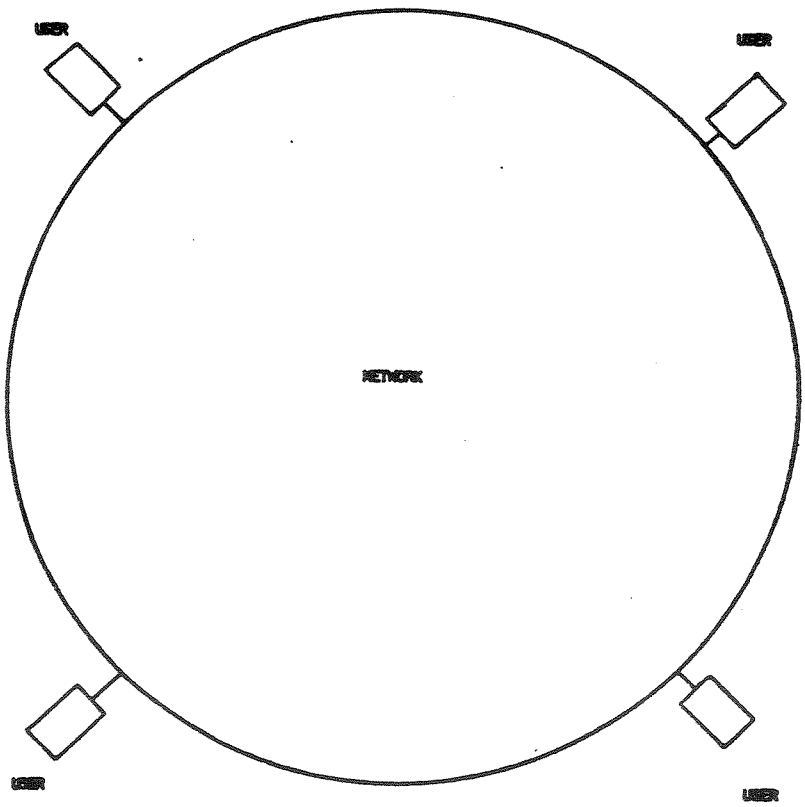

Figure 1. Network Top-Level Structure

separate, complete communication. Even at this early stage of development, the network can be written in Gypsy.

```
program Network(var upa:PortArray) = pending;

type PortArray = array(UserId) of Port;

type UserId = integer(1..NUsers);
const NUsers:integer = pending;

type Port = record(Get,Put:Line);
type Line = buffer(CSize) of Message;
const CSize:integer = pending;

type Message = pending;
```

This program gives a precise description of the lines of communication between the network and its external environment. Communication is through an "array(UserId) of Port." Each port is a record consisting of two buffers, and each buffer contains a maximum of csize messages. The type userid is an integer restricted to the range (1..nusers). The actual number of users, the maximal buffer size, the structure of messages, and the implementation of the network are left pending.

In a simple network with no protocols, the fundamental specification that is desired is that messages are delivered properly among all possible pairs of users. This specification for the network can be written as

```
program Network(var upa:PortArray) =
begin
  block all i,j:userid,
      ProperDelivery(i,j,upa);
  pending;
end;
```

The specification is written as a block, instead of an exit, specification because we intend the message network to be non-terminating. The network being blocked means that all processes in the network are blocked. This could happen for any number of reasons, including deadlock, but in this example, it will mean that there is no further input available from any user.

Before we can proceed with the implementation of the network, it is necessary that we be more specific about the meaning of "properdelivery." Loosely speaking, what we mean is that user j receives only those messages that were intended for j. We will make this definition precise with a macro.

```
define ProperDelivery(i,j,pa) =
```

```
    mail(pa(j).Put.outto,i,j)
        sub mail(pa(i).Get.infrom,i,j);
```

(The macro definition was chosen to illustrate the use of macros. ProperDelivery also could have been defined using a function.) Pa(j).put.outto is the sequence of all messages sent out to buffer pa(j).put by the network, and pa(i).get.infrom is the sequence of messages received in from buffer upa(i).get. The function mail(ms,i,j) is the subsequence of messages in message sequence ms that are directed from port i to j.

The completion of the definition of properdelivery requires a precise definition of the mail function, and mail in turn will require some additional information about messages.

```
function mail(ms:MessageSequence;i,j:UserId)
    :MessageSequence =
begin
  exit (assume mail(ms,i,j) =
    if ms=MessageSequence()
    then MessageSequence()
    else if i = Source(first(ms)) and
          j = Destination(first(ms))
      then MessageSequence(first(ms))
          @ Mail(nonfirst(ms),i,j)
      else Mail(nonfirst(ms),i,j)
    fi   fi);
end;

type MessageSequence = sequence of Message;

type Message<Source,Destination,Text,Compose,
    Equal> =
begin
  axiom all m:Message,
      Equal(Compose(Source(m),Destination(m),
          Text(m)),m);
  pending;
end;

function Source(m:Message):UserId =
    pending;
function Destination(m:Message):UserId =
    pending;
function Text(m:Message):CString =
    pending;
function Compose(s,d:Userid;t:CString)
    :Message = pending;
function Equal(m1,m2:Message):boolean =
    pending;
```

```
type CString = sequence (100) of char;
```

The definition of mail(ms,i,j) is given as an assumed exit specification which gives a complete recursive definition of mail. The definition of mail requires a new type, messagesequence. The type definition "sequence of message" defines a potentially infinite sequence of messages. Sequences are given a precise meaning by the semantics of Gypsy, but it is not necessary that they be implemented. Gypsy has a number of these kinds of constructs. They are included for purposes of formal program analysis, and may appear anywhere in a program where execution is not required, such as in specifications that are proved or assumed. In contrast, the type cstring is a sequence of ASCII characters of maximal size 100. Normally a Gypsy implementation would contain finite sequences but not infinite ones. Size restrictions can be enforced by run-time checks, and both kinds of sequences share a common semantics. Messagesequence() denotes the empty sequence of messages. In general, type names can be used to construct objects of that type. The @ operator is the sequence append operator.

The definition of mail makes use of two functions on messages, source and destination. The type definition of message permits these functions, as well as text, compose, and equal, access to the internal structure of messages, which is left pending. The axiom states an identity relation that must be maintained among this set of functions. This axiom implies that three kinds of information can be extracted from a message, a source, destination, and text part. The source and destination are the means of directing a message from one user to another, and the text is the actual content of the message to be transmitted. The compose function builds a message from these three parts, and equal defines a message equality. This is the only information that we will need to know about messages to carry out the full specification, implementation, and verification of the network process. Eventually, of course, we must choose a concrete representation of messages, and prove that the representation and the implementation of the functions that can access it satisfy the axioms.

Now we can give a completely precise interpretation to properdelivery. For every i,j pair, the mail from source i that is sent out to port j must be a subsequence of the mail received in from port i that is designated for destination j. This requires that the messages be the same and that they arrive in the same order that they were sent. The subsequence relation permits the network to drop messages. This is a concession to the reality of potentially unrecoverable transmission failures. This completes the specification of network.

We can proceed with the top-down design at any place in the current Gypsy program where a pending appears. There are many ways this program could be implemented to satisfy the block specification, but we will choose the following.

```
program network(var upa:PortArray) =
begin
  block all i,j:userid,
      ProperDelivery(i,j,upa);
  var npa:PortArray;
  cobegin
    Node(upa(i),npa(i),i) each i : UserId;
    switch(npa);
  end;
end;

process Node(var up,np:Port;i:UserId) =
begin
  block up.Put.outto sub np.Put.infrom
      and np.Get.outto sub up.Get.infrom;
  pending;
end;

process Switch(var npa:PortArray) =
begin
  block all i,j:UserId,
      ProperDelivery(i,j,npa);
  pending;
end;
```

This implements the program as a star network where each user is attached to exactly one node, and all of the nodes are connected to a single switch as shown in figure 2. Each node is similar to a full-duplex channel program passing messages unaltered, and in sequence, between the user and the central switch. All of the nodes and the switch are set into concurrent execution by the cobegin in the network program.

A node can be implemented by decomposing it into two one-way channels operating asychronously.

```
process Node(var up,np:Port;i:UserId) =
begin
  block up.Put.outto sub np.Put.infrom
    and np.Get.outto sub up.Get.infrom;
  cobegin
    Pass(np.Put,up.Put,i,Depart);
    Pass(up.Get,np.Get,i,Arrive);
  end;
end;
```

```
process Pass(var x,y:Line; i:UserId;
    d:Direction) =
begin
  block y.outto sub x.infrom;
  var m:Message;
  loop
    assert y.outto sub x.infrom;
    receive m from x;
    trace i = if d = Depart
        then Destination(m)
        else Source(m) fi;
    send m to y;
  end;
end;

type direction = (Arrive, Depart);
```

Pass is intentionally programmed as a non-termination loop. The loop simply receives messages from line x and passes them on to

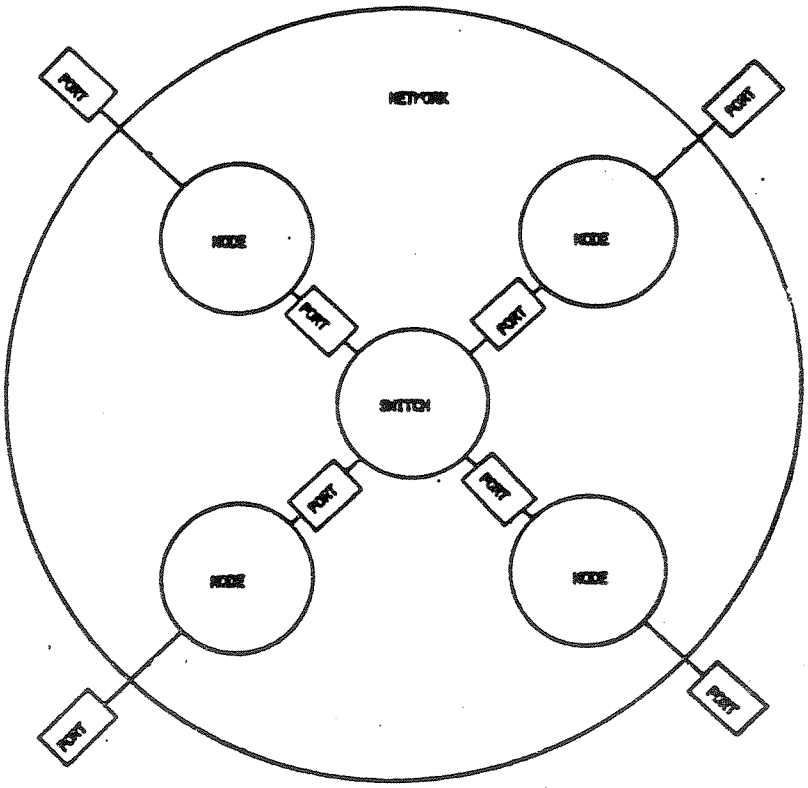

Figure 2. Network First-Level Refinement

line y performing a trace depending on the value of d. The send and receive statements are potential blockage points, and these are the points where the block specification must hold. A cobegin, as in node or network, also is a potential blockage point.

The switch process also loops forever waiting on each buffer in its turn for a small time slice. If input is ready it will receive it; otherwise, it will time out and go on to the next buffer.

```
process Switch(var npa:PortArray) =
begin
  block all i,j:UserId,
      ProperDelivery(i,j,npa);
  var m:Message;
  var k:UserId;
  cond DestinationErr;
  keep Destination(m) in (1..NUsers)
      otherwise DestinationErr;
  loop
    k := 1;
    loop
      if k > NUsers then leave end;
      assert all i,j:UserId,
          ProperDelivery(i,j,npa);
      await
      on receive m from npa(k).Get:
          send m to npa(destination(m)).Put;
      after TimeSlice: ;
      when
      is DestinationErr: ;
      end;
      k := k + 1;
    end;
  end;
end;

const TimeSlice:integer = pending;
```

Switch repeatedly iterates through the get buffers of the ports attempting to receive a message. Control leaves the inner loop at the leave statement, and the outer loop runs indefinitely. If a message is not received in timeslice amount of time, the await is exited and the next buffer is considered. If a message is received within the allocated amount of time, it is sent to the appropriate destination. The keep specification of switch is evaluated each time one of its variables is assigned a new value. If the specification ever is violated, a destination error is signalled. The keep prevents an invalid array index in the send

statement. If the error occurs, control is transferred to the when clause of the await and the destinationerr part of the when is performed. In this case, switch does nothing, thus dropping the message. This conforms with the subsequence relation specified in properdelivery.

The process structure of the complete network is shown in figure 3. All of these processes run concurrently. The intermediate level of a node process was not necessary. The pass processes could have been invoked explicitly from the cobegin in the network. The extra level of decomposition is helpful conceptually and in breaking the network into small, individually verifiable components.

Now let us return to the implementation of messages. They will be implemented in the obvious way as a record of three fields.

```
type Message<Source,Destination,Text,Compose,
    Equal> =
```

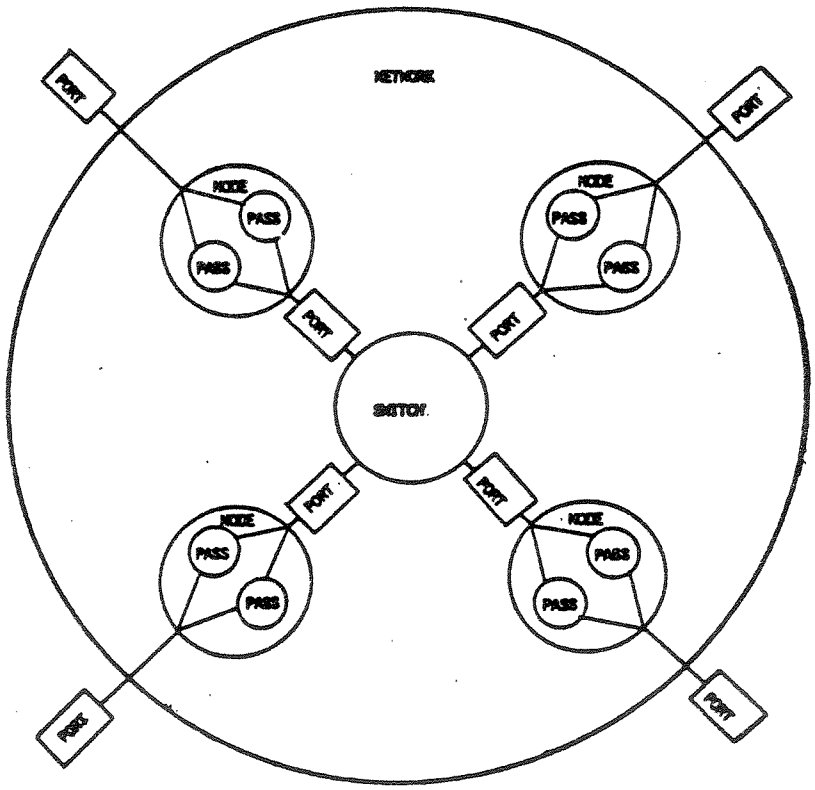

Figure 3. Network Second-Level Refinement

```
begin
  axiom all m:Message,
      Equal(Compose(Source(m),Destination(m),
          Text(m)),m);
  record(s,d:Userid; t:CString);
end;

function Source(m:Message):UserId =
begin
  cexit Source(m) = m.s;
  result := m.s;
end;

function Destination(m:Message):UserId =
begin
  cexit Destination(m) = m.d;
  result := m.d;
end;

function Text(m:Message):CString =
begin
  cexit Text(m) = m.t;
  result := m.t;
end;

function Compose(s,d:UserId; t:CString)
    :Message =
begin
  cexit Compose(s,d,t) = Message(s,d,t);
  result := Message(s,d,t);
end;

function Equal(m1,m2:Message):boolean =
begin
  exit Equal(m1,m2) iff Equal(m2,m1);
  cexit Equal(m1,m2) iff
      m1.s=m2.s and m1.d=m2.d and m1.t=m2.t;
  result := m1.s=m2.s and m1.d=m2.d
      and m1.t=m2.t;
end;
```

In the functions that are permitted access to the internal structure of messages, centry and cexit specifications also are permitted access to the internal structure, but entry and exit specifications are not. Entry and exit specifications are visible externally, to the routines that call the functions, but centry and cexit specifications are not. This prevents the external specifications from revealing the internal structure of messages. In a function the local variable with the reserved name "result" is the value assigned to the function upon exit.

It can be used in the same way as any other local variable. In the function compose, message(s,d,t) is another example of the type name used to construct an object of that type. Message(s,d,t) creates a message with successive fields equal to s, d, and t.

This completes the program and its specifications except for assigning values to the pending constants nusers, csize, and timeslice. The program at this stage of development can be written as

```
program network(var upa:PortArray) =
begin
  block all i,j:userid,
      ProperDelivery(i,j,upa);
  var npa:PortArray;
  cobegin
    Node(upa(i),npa(i),i) each i : UserId;
    switch(npa);
  end;
end;

type PortArray = array(UserId) of Port;

type UserId = integer(1..NUsers);
const NUsers:integer = pending;

type Port = record(Get,Put:Line);
type Line = buffer(CSize) of Message;
const CSize:integer = pending;

define ProperDelivery(i,j,pa) =
  mail(pa(j).put.outto,i,j)
      sub mail(pa(i).get.infrom,i,j);

function mail(ms:MessageSequence;i,j:UserId)
    :MessageSequence =
begin
  exit (assume mail(ms,i,j) =
    if ms=MessageSequence()
    then MessageSequence()
    else if i = Source(first(ms)) and
          j = Destination(first(ms))
      then MessageSequence(first(ms))
          @ Mail(nonfirst(ms),i,j)
      else Mail(nonfirst(ms),i,j)
   fi   fi);
end;

type MessageSequence = sequence of Message;
```

```
process Node(var up,np:Port;i:UserId) =
begin
  block up.Put.outto sub np.Put.infrom
    and np.Get.outto sub up.Get.infrom;
  cobegin
    Pass(np.Put,up.Put,i,Depart);
    Pass(up.Get,np.Get,i,Arrive);
  end;
end;

process Pass(var x,y:Line; i:UserId;
    d:Direction) =
begin
  block y.outto sub x.infrom;
  var m:Message;
  loop
    assert y.outto sub x.infrom;
    receive m from x;
    trace i = if d = Depart
        then Destination(m)
        else Source(m) fi;
    send m to y;
  end;
end;

type direction = (Arrive, Depart);

process Switch(var npa:PortArray) =
begin
  block all i,j:UserId,
      ProperDelivery(i,j,npa);
  var m:Message;
  var k:UserId;
  cond DestinationErr;
  keep Destination(m) in (1..NUsers)
       otherwise DestinationErr;
  loop
    k := 1;
    loop
      if k > NUsers then leave end;
      assert all i,j:UserId,
          ProperDelivery(i,j,npa);
      await
      on receive m from npa(k).Get:
          send m to npa(destination(m)).Put;
      after Timeslice: ;
      when
      is DestinationErr: ;
      end;
      k := k + 1;
    end;
```

```
    end;
  end;

  const TimeSlice:integer = pending;

  type Message<Source,Destination,Text,Compose,
      Equal> =
  begin
    axiom all m:Message,
      Equal(Compose(Source(m),Destination(m),
          Text(m)),m);
    record(s,d:Userid; t:CString);
  end;

  type CString = sequence (100) of char;

  function Source(m:Message):UserId =
  begin
    cexit Source(m) = m.s;
    result := m.s;
  end;

  function Destination(m:Message):UserId =
  begin
    cexit Destination(m) = m.d;
    result := m.d;
  end;

  function Text(m:Message):CString =
  begin
    cexit Text(m) = m.t;
    result := m.t;
  end;

  function Compose(s,d:UserId; t:CString)
      :Message =
  begin
    cexit Compose(s,d,t) = Message(s,d,t);
    result := Message(s,d,t);
  end;

  function Equal(m1,m2:Message):boolean =
  begin
    exit Equal(m1,m2) iff Equal(m2,m1);
    cexit Equal(m1,m2) iff
        m1.s=m2.s and m1.d=m2.d and m1.t=m2.t;
    result := m1.s=m2.s and m1.d=m2.d
        and m1.t=m2.t;
  end;
```

There are many details of Gypsy that this example does not illustrate, but the development of this program and its specifications provides a good overview of the philosophy and capabilities of the language.

Chapter 5

CONCLUSION

Gypsy has a number of important and distinctive aspects. It is a high-level language for general purpose computing that also supports the development of systems programs. It includes facilities for concurrency and timing, execution in imperfect run-time environments, and an access control mechanism. Gypsy includes extensive and powerful facilities for expressing functional specifications of its programs and of the units from which the program is structured. All constructs in Gypsy are verifiable either by formal proof or run-time validation. Run-time validation can be used effectively to reduce the size and complexity of the formal proofs. Facilities are provided for decomposing both routines and data into small, logically meaningful, units that can be verified independently. This modularity greatly enhances the practical feasibility of formal proofs. We believe that integrating these features smoothly into a common language is a significant step forward in the design of languages to support the systematic development of highly reliable computer programs.

Appendix A

SYNTAX

2.3

```
<symbol> ::= <letter>|<digit>|<score>|<prime>|<opcode>|
    <quote>|<blank>|<bracket>|<special>
<letter> ::= # upper/lower case ASCII letter #
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<score> ::= _
<prime> ::= '
<opcode> ::= +|-|*|/|<|>|@|=|.|,|;|:
<quote> ::= "
<blank> ::=
<bracket> ::= <|>|(|)|{|}
<special> ::= # all other  ASCII characters #
```

2.4

```
<token> ::= <id>|<number>|<character>|<string>|<comment>
<id> ::= <letter> {{<letterdigitscore>}* <letterdigit>}!
<number> ::= {<digit>}+
<character> ::= <prime> <symbol>
<string> ::= <quote> {<symbol>}* <quote>
<letterdigitscore> ::= <letterdigit>|<score>
<letterdigit> ::= <letter>|<digit>
<comment> ::= { #string not containing { or }# }
```

2.5

```
<type unit> ::= <"type" type header> =
    <"type" type body>|pending
<"type" type header> ::= type {<access list>}!
    <"type type name" id> {<formal type parameter list>}!
    {<access list>}!
<"type" type body> ::= <"type" type declaration>|
    begin {<type spec> ;}* <"type" type declaration> end|
    begin {<type spec> ;}* pending end
```

2.5.1

```
<"simple type" id> ::= rational|integer|character|
    <"scalar type" id>
<"scalar type" id> ::= boolean|<"scalar type name" id>
<"scalar" type declaration> ::= ( <"scalar value" id>
    {, <"scalar value" id>}* )
```

2.5.2

```
<"simple" value> ::= <"rational" value>|
    <"character" value>|<"scalar" value>
<"rational" value> ::= <"integer" value>|
```

```
        <"integer" value> / <"integer" value>
    <"integer" value> ::= <number>|- <number>
    <"character" value> ::= <character>|<"character value" id>
    <"scalar" value> ::= <"scalar value" id>
```

2.5.3

```
    <"simple" type declaration> ::= {<"simple type name" id>}!
        <"simple" range>
    <"simple" range> ::= ( <"simple" expression> ..
        <"simple" expression> )
```

2.5.4

```
    <"type" type reference> ::= <"type type name" id>|
        <"type type name" id> <"type" range>|<"type" range>|
        <"type type name" id> <actual type parameter list>
    <"type" formal type reference> ::=
        <"type" type reference>|<"type type name" id>
        <actual/formal type parameter list>
    <actual type parameter list> ::= ( <actual type parameter>
        {, <actual type parameter>}* )|{empty}
    <actual type parameter> ::= <expression>|{empty}
```

2.5.5

```
    <"record of typel...typen" type declaration> ::=
        record ( <"typel" field declaration>
        {; <"typei" field declaration>}* )
    <"type" field declaration> ::= <"type" field header>
        : <"type" type declaration>
    <"type" field header> ::= <"type field name" id>
        {, <"type field name" id>}*
```

2.5.5

```
    <"array typel of type2" type declaration> ::=
        array ( <"typel" type reference> ) of
        <"type2" type declaration>
```

2.5.6

```
    <"simple" composition> ::= <"simple type name" id>
        ( <"simple" expression> )|_ ( <"simple" expression> )
    <"record of typel...typen" composition> ::=
        <"record of typel...typen type name" id> (
        <"typel" component> {, <"typei" component>}* )|
        _ ( <"typel" component> {, <"typei" component>}* )
    <"array typel of type2" composition> ::=
        <"array typel of type2 type name" id> (
        <"type2" component> {, <"type2" component>}* )|
        _ ( <"type2" component> {, <"type2" component>}* )
    <"type" component> ::= {<"integer" expression> *:}!
        <"type" expression>
```

2.6

```
<constant unit> ::= <"type" constant header>
    {: <"type" type reference>}! = <"type" expression>|
    <"type" constant header> : <"type" type reference>
    = pending
<local constant> ::= const <"type constant name" id>
    {: <"type" type reference>}! = <"type" expression>|
    const <"type constant name" id> :
    <"type" type reference> = pending
<"type" constant header> ::= const {<access list>}!
    <"type constant name" id>
```

2.7

```
<local variable> ::= <"type" variable header>
    {: <"type" type reference>}! {:= <"type" expression>}!|
    <"type" variable header> : <"type" type reference>
    = pending
<"type" variable header> ::= var <"type variable name" id>
    {, <"type variable name" id>}*
```

2.7.1

```
<"type" variable reference> ::= <"type variable name" id>|
    <"record of ...type..." variable reference> .
    <"type field name" id>|
    <"array typel of type" variable reference> (
    <"typel" expression> )
```

2.8

```
<"type" expression> ::= <"type" value>|
    <"type" composition>|<"type constant name" id>|
    <"type" variable reference>|
    <"type" primed variable reference>|
    <"type" function call> ( <"type" expression> )
```

2.8.1

```
<"type" function call> ::= <"type function name" id>
    {<actual parameter list>}!
<actual parameter list> ::= ( <actual parameter>
    {, <actual parameter list>}* )
<actual parameter> ::= <expression>|<"condition name" id>
```

2.8.2

```
<"rational" expression> ::= plus\+ <"rational" expression>|
    minus\- <"rational" expression>|
    <"rational" expression> power\**
    <"rational" expression>|<"rational" expression>
    plus\+ <"rational" expression>|
    <"rational" expression> minus\-
    <"rational" expression>|<"rational" expression>
    times\* <"rational" expression>|
    <"rational" expression> divide\/
```

```
        <"rational" expression>|<"integer" expression>
        div\// <"integer" expression>|
        <"integer" expression> modulus\mod
        <"integer" expression>
```

2.8.2

```
    <"boolean" expression> ::= <"simple" expression> eq\=
        <"simple" expression>|<"simple" expression> ne\<>
        <"simple" expression>|<"compound" expression> eq\=
        <"compound" expression>|<"compound" expression> ne\<>
        <"compound" expression>|<"simple" expression> lt\<
        <"simple" expression>|<"simple" expression> le\<=
        <"simple" expression>|<"simple" expression> gt\>
        <"simple" expression>|<"simple" expression> ge\>=
        <"simple" expression>|<"simple" expression> in
        <"simple" range>
```

2.8.2

```
    <"boolean" expression> ::= not <"boolean" expression>|
        <"boolean" expression> or <"boolean" expression>|
        <"boolean" expression> and <"boolean" expression>|
        <"boolean" expression> imp\-> <"boolean" expression>|
        <"boolean" expression> iff\eqv <"boolean" expression>
```

2.8.2

```
    <"boolean" expression> ::= some <"type variable name" id>
        {, <"type variable name" id>}* :
        <"simple" type reference> , <"boolean" expression>|
        all <"type variable name" id>
        {, <"type variable name" id>}* :
        <"simple" type reference> , <"boolean" expression>
```

2.8.2

```
    <"type" expression> ::= if <"boolean" expression>
        then <"type" expression> else <"type" expression> fi
```

2.8.3

```
    <"simple" expression> ::= pred ( <"simple" expression ) |
        succ ( <"simple" expression> ) |
        min ( <"simple" expression> , <"simple" expression> ) |
        max ( <"simple" expression> , <"simple" expression> )
```

2.8.3

```
    <"simple" expression> ::= lower ( <"simple type name" id>
        ) |upper ( <"simple type name" id> )
```

2.8.3

```
    <"simple" expression> ::= scale ( <"integer" expression> ,
        <"simple" type reference> )
    <"integer" expression> ::= ord ( <"simple" expression> )
```

2.9

```
<statement list> ::= {<statement> ;}* {<statement>}!|
    pending|pending ;|;
<statement> ::= <simple statement>|<compound statement>
<simple statement> ::= <assignment statement>|
    <routine call statement>|<leave statement>|
    <buffer statement>|<clock statement>|
    <statement spec>|<signal statement>
<compound statement> ::= <if statement>|
    <case statement>|<loop statement>|
    <begin statement>|<cobegin statement>|<await statement>
<end clause> ::= end|<when clause>
```

2.9.1

```
<assignment statement> ::= <"type" variable reference> :=
    <"type" expression>
```

2.9.2

```
<routine call statement> ::= <"routine name" id>
    <actual parameter list>
```

2.9.3

```
<leave statement> ::= leave
```

2.9.4

```
<if statement> ::= if <"boolean" expression> then
    <statement list> {else <statement list>}! <end clause>
```

2.9.5

```
<case statement> ::= case <"simple" expression>
    {<"simple" is clause>}+ {<else clause>}! <end clause>
<"simple" is clause> ::= is <"simple" value>
    {, <"simple" value>}* : <statement list>
<else clause> ::= else : <statement list>
```

2.9.6

```
<loop statement> ::= loop <statement list> <end clause>
```

2.9.7

```
<begin statement> ::= begin <statement list> <end clause>
```

2.10

```
<routine unit> ::= <routine header> = <routine body>|
    <routine header> = pending
<routine header> ::= <procedure header>|<function header>|
    <process header>|<program header>
<routine body> ::= begin {<external spec> ;}!
    {<internal spec> ;}! {<local declaration> ;}*
    {<local spec> ;}! <statement list> <end clause>
<local declaration> ::= <local variable>|<local condition>|
    <local constant>|<local macro>|<local clock>
```

```
2.10.1
      <formal parameter> ::= {<limitation>}!
          <"type variable name" id> {,
          <"type variable name" id>}* :
          <"type" formal type reference>|
          cond <"condition name" id>  {, <"condition name" id>}*
      <limitation> ::= var|const|copy
      <formal parameter list> ::= ( <formal parameter>
          {; <formal parameter>}* )
      <actual/formal type parameter list> ::=
          ( <actual/formal type parameter>
          {, <actual/formal type parameter>}* )
      <actual/formal type parameter> ::=
          <actual type parameter>|<"type variable name" id> :

2.10.2
      <procedure header> ::= procedure {<access list>}!
          <"procedure name" id> <formal parameter list>

2.10.3
      <function header> ::= function {<access list>}!
          <"type function name" id> {<formal parameter list>}!
          : <"type" type reference>

2.10.4
      <program header> ::= program {<access list>}!
          <"program name" id> <formal parameter list>

2.11
      <Gypsy program> ::= {<unit> ;}+
      <unit> ::= <type unit>|<routine unit>|
           <const unit>|<macro unit>

2.11.2
      <access list> ::= <"unit name" id>
          {, <"unit name" id>}*

3.1.1
      <formal type parameter list> ::= ( <formal type parameter>
          {; <formal type parameter>}* )
      <formal type parameter> ::= <"simple variable name" id>
          {, <"simple variable name" id>}*
          : <"simple" formal type reference>

3.1.2
      <"sequence of type" type declaration> ::= sequence <size>
          of <"type" type declaration>
      <size> ::= ( <"integer" expression> )|{empty}

3.1.2
      <"set of type" type declaration> ::= set <size> of
```

```
        <"type" type declaration>

3.1.2
    <"bag of type" type declaration> ::= bag <size> of
        <"type" type declaration>

3.1.3
    <"list of type" composition> ::=
        {<"list of type type name" id>}! ( {<"type" component>
        {, <"type" component>}*}! )

3.1.3
    <"string" value> ::= <string>

3.1.4
    <"boolean" expression> ::= <"list" expression> eq\=
        <"list" expression>|<"list" expression> ne\<>
        <"list" expression>

3.1.4
    <"boolean" expression> ::= <"list" expression>
        sub <"list" expression>|<"type" expression>
        member <"list of type" expression>

3.1.4
    <"sequence" expression> ::= <"sequence" expression>
        append\@ <"sequence" expression>

3.1.4
    <"set\bag" expression> ::= <"set\bag" expression>
        union <"set\bag" expression>|<"set\bag" expression>
        intersect <"set\bag" expression>|<"set\bag" expression>
        difference <"set\bag" expression>

3.1.5
    <"type" expression> ::= first (
        <"sequence of type" expression> )|last (
        <"sequence of type" expression> )
    <"sequence" expression> ::= nonfirst (
        <"sequence" expression> )|nonlast (
        <"sequence" expression> )
    <"integer" expression> ::= length (
        <"sequence" expression> )

3.1.5
    <"boolean" expression> ::= ismerge (
        <"sequence" expression> ,
        <"bag of sequence" expression> )

3.2.1
    <"buffer of type" type declaration> ::= buffer <size> of
```

```
        <"type" type declaration>

3.2.2
    <buffer statement> ::= send <"type" expression> to
        <"buffer of type" variable reference>|receive
        <"type" variable reference> from
        <"buffer of type" variable reference>

3.2.4
    <"boolean" expression> ::= empty (
        <"buffer" variable reference> )|full (
        <"buffer" variable reference> )

3.2.5
    <process header> ::= process {<access list>}!
        <"process name" id> <formal parameter list>

3.2.6
    <cobegin statement> ::= cobegin
        <extended routine call statement>
        {; <extended routine call statement>}* <end clause>

3.2.7
    <extended routine call statement> ::=
        <routine call statement> {<each clause>}!
    <each clause> ::= each <"type variable name" id>
        : <"type" type reference>

3.2.8
    <await statement> ::= await {on <buffer statement> :
        <statement list>}+ {after <"integer" expression> :
        <statement list>}! <end clause>

3.2.9
    <local clock> ::= clock <"clock variable name" id>
        {, <"clock variable name" id>}*

3.2.10
    <clock statement> ::= log <"integer" variable reference>
        at <"clock variable name" id>

3.3
    <macro unit> ::= <macro header> = <"untyped" expression>
    <local macro> ::= define <"macro name" id>
        {( <"untyped variable name" id>
        {, <"untyped variable name" id>}* )}!
        = <"untyped" expression>
    <macro header> ::= define {<access list>}!
        <"macro name" id> {( <"untyped variable name" id>
        {, <"untyped variable name" id>}* )}!
```

3.6

```
<assertion> ::= <"boolean" expression>|
    <"boolean" expression> otherwise <condition>|
    ( <assertion clause> {; <assertion clause>}* )
<assertion clause> ::= prove <"boolean" expression>|
    prove <"boolean" expression> otherwise <condition>|
    assume <"boolean" expression>|
    assume <"boolean" expression> otherwise <condition>
<"type" primed variable reference> ::=
    <"type" variable reference> '
```

3.6.1

```
<type spec> ::= {require <assertion>}!
    {axiom <assertion>}!
```

3.6.2

```
<statement spec> ::= assert <assertion>
    trace <"boolean" expression>
```

3.6.3

```
<external spec> ::= {<entry spec>}! {<block spec>}!
    {<exit spec>}!
<entry spec> ::= entry <assertion>
<block spec> ::= block <assertion>
<exit spec> ::= exit <assertion>|exit case
    ( {is <condition> {, <condition>}* : {<assertion>}!}+ )
<internal spec> ::= {<centry spec>}! {<cblock spec>}!
    {<cexit spec>}!
<centry spec> ::= centry <assertion>
<cblock spec> ::= cblock <assertion>
<cexit spec> ::= cexit <assertion>|cexit case
    ( {is <condition> {, <condition>}* : {<assertion>}!}+ )
<local spec> ::= keep <assertion>
```

3.7.1

```
<local condition> ::= cond <"condition name" id>
    {, <"condition name" id>}*
<condition> ::= <"condition name" id>|indexerror|caseerror|
    aritherror|initializeerror|buffererror|routineerror|
    rangeerror|senderror|receiveerror|assignerror|typeerror
```

3.7.2

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

3.7.3

```
<when clause> ::= when {is <condition> {, <condition>}*
    : <statement list>}+ {else : <statement list>}! end
```

Appendix B

STANDARD OPERATORS

	OPERATOR	OPERAND1	OPERAND2	OPERAND3	RESULT	PRIORITY	RUN-TIME
power	**	r	r		r	1	yn
plus	+		r		r	2	y
minus	-		r		r	2	y
times	*	r	r		r	3	y
divide	/	r	r		r	3	n
div	//	i	i		i	3	y
modulus	mod	i	i		i	3	y
plus	+	r	r		r	4	y
minus	-	r	r		r	4	y
append	@	sq	sq		sq	4	y
union		sb	sb		sb	4	y
intersect		sb	sb		sb	4	y
difference		sb	sb		sb	4	y
eq	=	eb	eb		b	5	y
ne	<>	eb	eb		b	5	y
lt	<	s	s		b	5	y
le	<=	s	s		b	5	y
gt	>	s	s		b	5	y
ge	<=	s	s		b	5	y
in		s	sr		b	5	y
sub		l	l		l	5	y
member		l	l		l	5	y
not			b		b	6	y
and		b	b		b	7	y
or		b	b		b	8	y
imp	->	b	b		b	9	y
iff	eqv	b	b		b	9	y
some		s	bs		s	10	n
all		s	bs		s	10	n
if...fi		b	eb	eb	eb		y

Where "r" = rational, "i" = integer, "s" = simple, "b" = boolean, "sq" = sequence, "sb" = set/bag, "eb" = except buffer, "sr" = simple range, "l" = list, "bs" = boolean(simple), and "yn" = yes, but only when the exponent is a positive integer.

Appendix C

RESERVED WORDS

after 3.2.8
all 2.8.2
allfrom 3.2.3
allto 3.2.3
and 2.8.2
append 3.1.4
aritherror 3.7.1
array 2.5.5
assert 3.6.2
assignerror 3.7.1
assume 3.6
at 3.2.10
await 3.2.8
axiom 3.6.1
bag 3.1.2
begin 2.5, 2.9.7, 2.10
block 3.6.3
boolean 2.5.1
buffer 3.2.1
buffererror 3.7.1
bufq 3.2.3
case 2.9.5, 3.6.3
caseerror 3.7.1
cblock 3.6.3
centry 3.6.3
cexit 3.6.3
character 2.5.1
clock 3.2.9
cobegin 3.2.6
cond 2.10.1
const 2.6, 2.10.1
copy 2.10.1
define 3.3
difference 3.1.4
div 2.8.2
divide 2.8.2
each 3.2.7
else 2.8.2, 2.9.4, 2.9.5, 3.7.3
empty 3.2.4
end 2.5, 2.9, 3.7.3
entry 3.6.3
eq 2.8.2, 3.1.4
eqv 2.8.2
exit 3.6.3
fi 2.8.2
first 3.1.5
from 3.2.2
full 3.2.4
function 2.10.3
ge 2.8.2
gt 2.8.2
if 2.8.2, 2.9.4
iff 2.8.2
imp 2.8.2
in 2.8.2
indexerror 3.7.1
infrom 3.2.3
initializeerror 3.7.1
integer 2.5.1
intersect 3.1.4
is 2.9.5, 3.6.3, 3.7.3
ismerge 3.1.5
keep 3.6.3
last 3.1.5
le 2.8.2
leave 2.9.3
length 3.1.5
log 3.2.10
loop 2.9.6
lower 2.8.3
lt 2.8.2
max 2.8.3
member 3.1.4
min 2.8.3
minus 2.8.2
mod 2.8.2
modulus 2.8.2
ne 2.8.2, 3.1.4
nonfirst 3.1.5
nonlast 3.1.5
normal 3.6.3
not 2.8.2
of 2.5.5, 3.1.2, 3.2.1
on 3.2.8
or 2.8.2
ord 2.8.3
otherwise 3.6
outto 3.2.3

Bibliography

[1] Brinch Hansen, Per. "The Nucleus of a Multiprogramming System," CACM 13, 4 (1970).

[2] Brinch Hansen, Per. "Operating Systems Principles," Prentice-Hall (1973).

[3] Brinch Hansen, Per. "The Purpose of Concurrent Pascal," Proceedings ICRS (1975).

[4] Burger, Wilhelm. "Formal Semantic Definition of GYPSY", (in preparation).

[5] Buxton, J.N. and B. Randell, eds. Software Engineering Techniques, NATO Science Committee (1970).

[6] Clint, M. "Program Proving: Co-routines," Acta Informatica, 2 (1973).

[7] Dahl, O. -J. "Notes on Data Structuring," Dahl, Dijkstra, and Hoare, Structured Programming, Academic Press (1972).

[8] Dijkstra, Edsger W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," CACM 18, 8 (1975).

[9] Flon, Lawrence. "A Survey of Some Issues Concerning Abstract Data Types," Technical Report, Carnegie-Mellon (1974).

[10] Good, D.I., and L.C. Ragland. "Nucleus--A Language for Provable Programs," Program Test Methods, Hetzel (ed.), Prentice-Hall (1973).

[11] Good, D.I., et al. "Gypsy Compiler Directive," (in preparation).

[12] Igarashi, S., R.L. London, and D.C. Luckham. "Automatic Program Verification I: A Logical Basis and Its Implementation," Report ISI/RR-73-11, Inf. Sci. Inst. USC (1973).

[13] Jensen, Kathleen and Niklaus Wirth. "Pascal User Manual and Report," Springer Verlag (1974).

[14] Lampson, B.W. et al. "Euclid Report," Draft (1976).

[15] Liskov, Barbara and Stephen Zilles. "An Approach to Abstraction," Computation Structures Group Memo 88, MIT (1973).

[16] Lyle, Don M. "A Hierarchy of High Order Languages for Systems Programming," Proc. of SIGPLAN Symp. on Languages for Systems Implementation (1971).

[17] Moriconi, Mark S. "An Interactive System for Incremental Program Design and Verification," (in preparation).

[18] Randall, B., "System Structure for Software Fault Tolerance," Proceedings ICRS (1975).

[19] Stucki, L.G. "Testing Impact on the Future of Software Engineering," Proc. of Fourth Texas Conf. on Computing Systems, University of Texas (1975).

[20] van Wijngaarden, A., et al. "Report on the Algorithmic Language ALGOL 68," Numerische Mathematik, 14 (1969).

[21] Wulf, W.A., D.B. Russell, and A.N. Habermann. "BLISS: A Language for Systems Programming," CACM 14, 12 (1971).

[22] Wulf, W., R. Levin, and C. Pierson. "Overview of the Hydra Operating System Development," Proc. of Fifth Symp. on Operating Systems Principles, (1975).

[23] Wulf, W.A., R.L. London, and Mary Shaw. "Abstraction and Verification in Alphard", preliminary draft, Carnegie-Mellon (1976).

[24] Zahn, Charles T., "A Control Statement for Natural Top-Down Structured Programming," Symp. on Programming Languages (1974).

Index