# A Gypsy-to-Ada Program Compiler

**Robert L. Akers**

**December 1983 Technical Report 39**

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

# Table of Contents

# Acknowledgements

Don Good, Rich Cohen, Larry Smith, and Mike Smith assisted in the selection of the compiler as a thesis topic. Don Good was primarily responsible for acquiring funding for the project.

Bill Young assisted in the implementation of dynamic structured types by coding the Ada support packages of standard functions for those types. Discussions with Bill Young and Don Good were invaluable in arriving at the final design for the implementation of dynamic types. John McHugh made significant contributions in discussions on the implementation of condition handling.

Larry Smith was my constant sounding board. His contribution cannot be underestimated, nor my appreciation sufficiently expressed.

All of these people have made valuable suggestions and provided numerous insights in discussions which helped formulate the implementation.

# Chapter 1
# INTRODUCTION

Gypsy [11] is a language developed to support a methodology for formally specifying, implementing, and verifying computer software. Developed from Pascal [18,35], it is a concise, clean, and very modular language in the lineage rooted in ALGOL 60 [23]. The evolution of Gypsy from Pascal was largely a process of eliminating those language constructs which did not lend themselves to verifiability, replacing them with more tractable alternatives, and adding some new capabilities, including a mechanism for the formal specification of program behavior.

The Gypsy Verification Environment [6,7,8] is an interactive system of tools, implemented in LISP, which work together to aid the programmer in executing the tasks of incremental program development and verification. Programs may be incrementally specified, implemented, parsed, proven, and compiled, all within this environment.

Ada [16,32] is a programming language of considerable expressive power designed to be applicable to embedded systems and a wide domain of other applications. Ada also has its roots in the heritage of ALGOL and Pascal, and bears many syntactic and semantic resemblances to Gypsy, though Ada is much larger and more complex. The critical nature of some of its applications has aroused interest in the possibility of verified Ada programs, but certain aspects of the language preclude of formal verification [31,37].

The subject of this report is the design and initial implementation of a Gypsy compiler which targets to Ada. The compiler is the combination of a LISP module integrated into the Gypsy Verification Environment and packages of support routines written in Ada. The compiler converts an internal form of a Gypsy program maintained by the verification system into Ada source, which may then be compiled along with the support packages in any Ada environment. Using the Gypsy verification methodology and tools to perform the actual task of program verification and then applying the compiler to obtain Ada code with the same semantics results in Ada code which may be used with much the same confidence as the verified Gypsy program.

There are several aspects of this project, some pragmatic and some theoretical, which are attractive to those interested in formal program verification. First, this approach will require minimal time and expense in producing verified Ada, since the tools it utilizes for program development and verification already exist and have demonstrated success. Such a system of tools is normally huge and costly, and producing a whole system tailored to Ada might be prohibitively expensive. Secondly, this approach will provide a mechanism for getting Gypsy programs to run on any machine which can run Ada. Thirdly, this approach effectively decouples the development of Gypsy and Ada. To users, this means that Gypsy is a verifiable systems development language which can compile into Ada. To the verification research community, this means verification research may be cleanly decoupled from Ada. The importance of this decoupling is that it allows subsequent development of Gypsy and Ada to proceed without mutual interference.

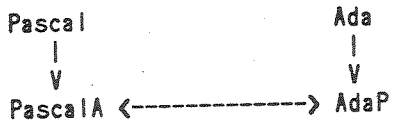
This report will briefly outline background to the compiler effort, describe the mapping from Gypsy to Ada constructs, and describe the role of the compiler in the Gypsy Verification Environment and the high points of its implementation.

## 1.1 Background -- Compiling to High Level Languages

Usually, programs are compiled from a high level language to some machine language, but there are often reasons for compiling programs from one high level language into another. For example, an enterprise may wish to consolidate its software effort to a minimum number of languages so as to streamline and simplify its personnel assignments. Two other cases, the ones motivating this report, are: 1) that special program development strategies available for one language do not exist for the language in which the program must ultimately be compiled, and 2) that a program written in some language may be needed on another machine for which no compiler for the language exists.

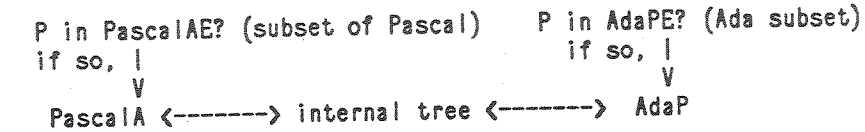
Source-to-source compiling from one high level language to another is not nearly as common as compiling to a lower level language, nor has it received as much attention in the literature. Nevertheless, it is not without precedent. One well documented effort, which is germane to this report, is the Pascal-to-Ada and Ada-to-Pascal translator implemented by the PascAda group at the University of California at Berkeley [2]. The problems of translating from Pascal to Ada are quite similar to those of translating Gypsy to Ada. (The project documented in this report is not concerned with problems analogous to compressing features of the Ada language into the less baroque Pascal. The questions of reflexive translation, whereby a program translated from Gypsy to Ada and back to Gypsy needs to resemble its original form, is never raised here.)

The central strategy of the PascAda group was to define compatible subsets of Pascal and Ada which have a direct translation between them. A program written in full Pascal is transformed into a form using only those constructs in the Pascal subset. The program in Pascal subset form is then converted into the Ada subset form and regenerated as Ada source for interpretation or compilation. The converse process is followed for translation from Ada to Pascal. The target, then, is the subset language rather than the full version of the target language. Where AdaP is the Ada subset and PascalA the Pascal subset, the general schema of the translation is:




While PascalA and AdaP are semantically equal, primarily syntactic issues will make the incarnations of a single program look somewhat different in the two languages. To aid in the transition, a single intermediate form was selected to represent the semantics of a program in either language. This form could be used as a basis for generating the syntactically appropriate source code in either language. Translations to and from the intermediate form are generally local in nature. The intermediate form chosen was the tree structure defined in the formal definition of Ada [9].

Some features of the each language have no analog at all in the other language, and these were thereby excluded from translatability. (This is primarily true for Ada features which do not map into the simpler Pascal.) A translatable subset, larger than the semantically equal subsets, was defined for each language. These were called AdaPE for the Ada side and PascalAE for Pascal. The process of translating a program P, say from Ada to Pascal, then became: 1) determining if P is in AdaPE (is translatable), 2) transforming P (an AdaPE program) into AdaP, 3) translating the AdaP representation of the program into the intermediate tree structure, and 4) generating the PascalA program from the tree. Schematically the arrangement is:




Since the intermediate tree form was designed to represent Ada programs, the algorithm to prettyprint the Ada source is a straightforward, top-down, post order traversal, which prints text as it goes. (This is quite similar to the operation of infprint on Gypsy prefix representations in the Gypsy Verification Environment.) Since the tree representation was designed for Ada, some information often needed to be collected or inferred in order to print Pascal from the tree. (Similar collection is necessary in generating Ada from the modified prefix which serves as the translation medium for the Gypsy-to-Ada compiler.)

## 1.2 Goals And Priorities

The implementation of the Gypsy to Ada compiler is an ongoing project. The work documented here is the attempt to compile the bulk of the Gypsy language, with the notable exception of its mechanisms for concurrency. It may be reasonably expected that work on concurrency may follow, but our goal has been to compile accurately the large subset of Gypsy which does not involve concurrency before expanding to the full language.

We realized that on occasion there would be certain uses of Gypsy language features which would not lend themselves at all to the compilation effort, or for which the overhead of compilation would exceed the benefits. The compilation philosophy used in this report allows for the acceptability of variances where such cases might arise.

The primary role of the compiler is simply to aid in producing running object code from Gypsy programs, which may have been completely verified. It need consider, then, only those parts of the program which will form its runtime image. All code which contributes only to the verification process or otherwise to program development may be disregarded. To the greatest extent feasible, remnants of untranslated Gypsy text, for example non-validated formal specifications and lemmas, are maintained as comments in the Ada code.

Our approach concerns only the one-way compilation from Gypsy to Ada. The task of compiling Ada programs into Gypsy is much more difficult, due to the generality of Ada. Some of Ada's fundamental characteristics which do not lend themselves to verification might likewise not lend themselves to representation in a language which is restricted by the rigors of verifiability.

It is hoped that the results of this effort will serve as fruitful ground for the further study of language-related issues regarding Gypsy and its usage, as well as its relation to Ada.

Clearly, the foremost priority of the compiler is that the Ada program must be functionally equivalent to the Gypsy program so as to preserve the validity of the Gypsy proofs. A secondary priority is to maintain a physical resemblance of the Ada program to the original Gypsy. It is understood that this resemblance must dissolve in the compilation of many constructs which are treated differently in Ada, even though the differences may be subtle. A less significant priority is the efficiency of the Ada program.

## 1.3 Caveat On Ada Source Modifications

Since the validity of the verification of a Gypsy program is extremely sensitive to changes in the program, the Gypsy Verification Environment (GVE) includes a facility for tracking the effects of such changes through incremental program development. This facility notes which verification conditions will have to be regenerated after a change and which unchanged ones will have to be re-proven in order to maintain the validity of the verification. Changes occurring outside the GVE impose a complete caveat emptor on the verification status of the program.

A verified program may undergo several translations, all of which are designed to preserve verification status. These translations, supervised in the GVE, may include compilation via translation to BLISS for later BLISS compilation, or the translation to Ada described here. The role of any Gypsy compiler is to produce a functionally equivalent program in its target language. A program emitted from a compiler may be considered to be "verified" only if the Gypsy program being translated was completely verified in the GVE and was unchanged since the verification was performed. Similarly, the code produced by a compiler, be it BLISS or Ada, must not be altered. To do so would invalidate the verification of the program.

# Chapter 2
# THE ADA IMAGE OF GYPSY CONSTRUCTS

This chapter describes the Ada forms into which the various Gypsy constructs are compiled. It parallels the text of the Revised Report on the Language Gypsy Version 2.1, by Donald I. Good, [38] which is the current defining document for Gypsy. The treatment of Chapters 1-7 is the most thorough, since the compiler was designed first to completely translate the features described in those chapters. Material from the other chapters is covered to the extent to which the compiler software currently treats it.

## 2.1 Basic Concepts

The broad concept of a program in the two languages is identical, although Gypsy buffer parameters to the main procedure are treated directly as files in the Ada environment. The fundamental concept of procedure, function, type, and constant are similar in Gypsy and Ada. Gypsy lemmas, since they have no runtime significance, may be transformed into comments. Dynamic memory is implemented with explicit pointers, although this implementation is totally abstracted from user control in Gypsy.

The Gypsy standard types boolean, integer, and character are present in Ada, and the set of predefined operations for objects of those types are similar, but not identical. Where there are differences, Ada support code has been written to perform exactly as would the Gypsy program. Rational numbers, as currently implemented in Gypsy, serve no runtime purpose, so they are not translated. The type activationid is not relevant to the Ada image as currently implemented. Scalar types are supported as enumeration types in Ada, and additional support provided in the compiler ensures that their implementation reflects Gypsy semantics. Ada includes the standard composition types array and record. Ada's access types are used in conjunction with records to implement Gypsy sets, sequences, mappings, and buffers. Strings are present in Ada, and string constants in Gypsy and Ada share the same notation. Ada's predefined strings are implemented as arrays of character, however, while Gypsy's are sequences of character. The compiler implementation must treat them as it does sequences, which includes coercing constants from array to sequence (linked record) value.

Gypsy's standard procedures are emulated in three ways: 1) as routines in instantiations of predefined generic support packages, 2) as routines in a non-generic support package, or 3) with in-line expansions possibly incorporating calls to the above types of support routines. Gypsy's IF, CASE, and LOOP statements have counterparts in Ada with which the Gypsy semantics may be expressed.

The implementation preludes (see Ch. 2.5.9.3) which will be available for use with the Ada compiler will mainly provide support for the input/output mechanisms. They will include type and object representations which will represent internal objects eligible for treatment as input/output devices. When the user invokes the compiler from the GVE monitor, he will supply a Gypsy-like call to the user's main procedure which will link the program to an environment and drive the compiler's treatment of input/output.

## 2.2 Lexical Preliminaries

The character set available to Ada programs is the same as that for Gypsy. Ada identifiers have the same form as Gypsy identifiers. The bound on identifier length is an Ada implementation constraint. The set of reserved words is somewhat different in the two languages. Where a Gypsy identifier is the same as an Ada reserved word, it is transformed through translation to some non-reserved identifier. The set of identifiers beginning with the characters "g_a_" are reserved for use by the compiler.

Any construct translated as a comment becomes an Ada end-of-line comment, textually preceded by "--" on every line. A <CRLF> is inserted at the end of the comment.

## 2.3 Type Specifications

The concept of type is quite similar in Ada and Gypsy. The syntactic form of type specifications in Ada is quite similar to Gypsy's, and all varieties of Ada types are present in Gypsy with the exception of the access type. The fundamental target mechanisms for translating Gypsy types, then, are present in Ada. Even so, the translation of types and their predefined operations and the provisions made for replicating Gypsy type compatibility in Ada is one of the most involved processes undertaken in the compiler.

While both Ada and Gypsy are strongly typed languages, Ada places constraints on type compatibility which are not present in Gypsy. Type compatibility in Gypsy means only that the actual parameters in function and procedure calls must be of the same basetype as the formal parameters. (The same treatment is given to arguments to the various predefined statements.) Type compatibility is all that is required by the Gypsy parser for type checking. Checks on range and size restriction violations on the types of formals must commonly be deferred until runtime, when a violation will result in the signalling of the VALUEERROR condition.

In Ada, however, type compatibility means that the objects are either of the same declared type or are of explicitly declared subtypes which share the same basetype. For example, consider:

```
type int1 is integer;
type int2 is integer;
type int3 is integer range 1 .. 10;
```

Objects of any pair of these types are not type compatible with one another, since each type declaration represents a distinct data type. In order for them to be compatible, each must be declared as a subtype of type integer. Consider:

```
subtype int1 is integer;
subtype int3 is integer range 1 .. 10;
```

Objects of types int1 and int3 are type compatible both with each other and with objects declared to be of predefined type integer.

To achieve the Gypsy sense of type compatibility in the Ada program, then, it is necessary to declare explicitly the basetype of every type declared in the Gypsy program, and to declare each user-declared type to be a subtype of the appropriate basetype. Moreover, any objects sharing the same Gypsy basetype need to be linked to the same Ada basetype. Each basetype is declared only once in the Ada program. Consider the Gypsy type declarations in scope foo:

```
type array1 = array ([1..10]) of integer[1..10];
type array2 = array ([1..10]) of integer[0..1000];
```

These two array types will share the same basetype, which will be declared in BASETYPE_PACKAGE to be:

```
type array1_foo is array (integer range 1..10) of integer;
```

Both array1 and array2 are declared to be subtypes of array1_foo.

A further rationale for the use of this basetype mechanism relates to predefined operations in

Gypsy. These include predefined operators, standard functions, and predefined statement forms such as assignment. In many cases Gypsy provides operations which Ada does not, and in many other cases the operations provided have different semantics than their predefined Ada counterparts. The compiler must provide Ada support semantically equivalent to that of Gypsy. Strong typing in Ada suggests that these operations would be most efficiently defined for the various basetypes which occur in the program. (The Ada generic capability is not general enough to provide, for example, a single assignment procedure for all objects, but it may be used to provide a single assignment procedure for all integer types.) Any operation properly defined on the basetype of its operands may be used with operands of any subtype, provided a mechanism exists for providing appropriate behavior regarding subtype restrictions.

This mechanism involves the use of a typedescriptor. Each Gypsy type has associated with it a typedescriptor constant which describes it in terms of its kind (i.e. sequence, set, array, etc.), its range and size restrictions, and typedescriptors of its component types. A typedescriptor will be provided as a parameter to all functions and procedures which must be sensitive to range and size restrictions, for example where runtime valueerror or indexerror checking is performed. The typedescriptor type is declared in the Ada program to be a variant record, with the kind of Gypsy object as its discriminant. For an example of a typedescriptor, consider these Gypsy type declarations:

```
type smallpos = integer [1..100];
type small_array = array ([1..10]) of smallpos;
```

The typedescriptor declarations which would accompany the translations of these types are:

```
smallpos_typedescriptor: constant gypsy_package.typedescriptor
      := (kind => g_integer; low => 1; high => 10);

small_array_typedescriptor: constant
      gypsy_package.typedescriptor := (kind => g_array;
          index_type => (kind => g_integer; low => 1;
          high => 10); elem_type => smallpos_typedescriptor);
```

For a declaration of the typedescriptor type, see Appendix A.

The predefined operations on basetypes are provided in four different ways, depending on which is most appropriate. If the correct operation exists in the predefined Ada support, for example with integer comparisons, it may be used in its native form. The remaining three alternatives are generated during translation. For basetypes which are predefined in the compiler, i.e. integer, character, boolean, sequence of integer, sequence of character, and sequence of boolean, the predefined operations are provided in a support package named GYPSY_PACKAGE. For those other basetypes which result from user declared types, the functions are provided either through instantiation of generic packages of routines germane to a kind of type (array, sequence, set, mapping, buffer), or, where the routines do not lend themselves to generic instantiation, by in-line declaration based on templates embedded in the compiler. This latter method is used most notably on enumerated scalar and record types, which are n-ary in nature, and on functions supporting the seqconstructor and setconstructor operations. The listings for GYPSY_PACKAGE and the generic basetype packages are provided in Appendix A and Appendix B.

A feature of Ada very unfortunate for this project is that any program unit or object must be declared before it may be referenced. Since Gypsy scopes are translated into Ada packages and Ada packages must include in their headers the names of all other packages which they reference, there is a constraint on Gypsy input disallowing scopes which form a ring of inter-scope references via name declarations. This can create difficulties for programs not composed with this constraint in mind. Declaring a basetype in the same package as one of its subtypes, and then providing access to that package for all other packages using any other subtype of the basetype could greatly magnify the difficulty.

To simplify this problem, the compiler places all basetype declarations in a separate package, BASETYPE_PACKAGE, which appears first in the translated code. All other packages are declared with access to the basetype package. The basetype package also includes a valueerror checking function, default initial value and typedescriptor constant declarations, and the declarations of all the standard functions provided by Gypsy on objects of each basetype. Grouping all basetype declarations and their associated unit declarations in an isolated part of the translated program has

the desirable side effect of keeping directly translated code free of constructs radically different in appearance from the original Gypsy program.

For an example of a basetype package resulting from translation of a sample Gypsy program, see Appendix D.3.

### 2.3.1 Initial Values

Types in Gypsy have default initial values associated with them, while types in Ada do not. To provide an initial value, each type declared in translated code is accompanied by a constant declaration which specifies the initial value for the type. If the user has provided an initial value in his Gypsy type specifications, then that value is the one given as the value of the constant. Otherwise, the Gypsy default initial value for the type is used. Composed types have initial values based on the default initial values of their component types. Array and record initial values may, but need not, be expressed as aggregates. Dynamic types must have their initial values expressed in terms of calls to predefined functions which construct objects of the type. The default initial value for a dynamic object is a record structure with null item pointers.

For example, consider the following type declarations.

```
type foo_int1 = integer [1..10];
type foo_int2 = integer [1..100] := 100;
type foo_array = array ([1..10]) of foo_int1;
type foo_seq = sequence of integer;
```

The corresponding initial value constant declarations would be:

```
g_a_foo_int1_initial: constant foo_int1 := 1;
g_a_foo_int2_initial: constant foo_int2 := 100;
g_a_foo_array_initial: constant foo_array := (1..10 => 1);
g_a_foo_seq_initial: constant foo_seq :=
     basetype_package.foo_seq_basetype.null_value;
```

These constant declarations appear with the type declarations and may be used in any package which may refer to the type name.

All Gypsy variables are assigned initial values either explicitly in declarations or implicitly from the default initial value of the type. Their Ada counterparts must be given initial values explicitly. Consider the following declarations.

```
foo1: foo_int1;
foo2: foo_int2 := 10;
foo3: integer;
```

These will be translated to:

```
foo1: foo_int1 := g_a_foo_int1_initial;
foo2: foo_int2 := 10;
foo3: integer := 0;
```

### 2.3.2 Simple Types

The Gypsy simple types boolean, character, integer, enumerated scalar types, and subrange types are all provided in Ada, but some operations on objects of these types must be treated differently, and thus generated by the compiler. The operations eq, ne, lt, le, gt, ge, max, min must be defined for all simple types. The functions lower and upper must be defined for all bounded simple types. An assignment procedure must also be provided which responds to bounds violations by raising a valueerror rather than the Ada constraint_error.

Eq, ne, lt, le, gt, and ge are all defined in Ada and may be used in their infix form. The functions max and min are defined in GYPSY_PACKAGE as generic functions operating on range operands. Where T is a typename, lower(T) and upper(T) may be transformed into Ada's functional attributes T'first and T'last. Assignment is generated in a manner appropriate for each kind of type.

### 2.3.3 Scalar Types

The basetype of a scalar type is the scalar type itself. Since there may be any number of items in the value set, the type declaration may not be parameterized for generic treatment. Therefore, scalar type declarations are generated from templates and appear in BASETYPE_PACKAGE. The operations pred, succ, ord, and scale must be defined, and they, too, are generated from templates, along with the assignment procedure and valueerror checking function.

The typedescriptor for an enumerated scalar type is given in terms of the ordinal position of the lower and upper bounds of the type within the sequence of scalar identifiers composing the type.

#### 2.3.3.1 Type Boolean

Gypsy operations defined for type boolean are: not, and, or, imp, and iff. Of these, not, and, and or are sufficiently expressed in Ada. Since type boolean is one of the predefined basetypes known to the compiler, imp, iff, assignment, and the valueerror function are defined in GYPSY_PACKAGE. The universal and existential quantification operations are not intended for execution, so they are not translated. If they are detected in executable code, the translation fails. In non-executable code, they are printed in their original form for documentation purposes.

#### 2.3.3.2 Type Character

The type character is predefined in Ada and carries the same operations as scalar types. Since character is one of the predefined basetypes known to the compiler, the pred, succ, ord, and scale functions and the assignment procedure are declared in GYPSY_PACKAGE. Character literals are printed in the form consistent with the Ada declarations of the CHARACTER and ASCII packages. Printable characters are printed using the quoted form; control characters are printed using the constant identifiers from the ASCII package.

#### 2.3.3.3 Type Integer

The Gypsy operations on integers include "+", "-", "*", "**", div, mod, and unary "-". Ada supports all of these operations (Ada "rem", not "mod", is the Gypsy "mod".), but the only exceptions which can be raised by the Ada operators are CONSTRAINT_ERROR and NUMERIC_ERROR. Contrast this with the more specific conditions raised by the Gypsy operators, which include adderror, subtracterror, multiplyerror, powererror, powerindeterminate, negativeexponent, divideerror, zerodivide, and minuserror. So that the proper exceptions will be raised by the integer operations, the compiler provides each of the operations as a predefined function. Typically, these functions will have the call to the appropriate Ada operation embedded and will trap Ada exceptions with the appropriate granularity, converting them to the various Gypsy conditions. The functions, along with other declarations germane to integer support, are defined in GYPSY_PACKAGE. Note that they are declared as functions and not as overloaded operators. This means the infix operator forms will be translated into function calls with normal syntax.

Gypsy 2.1, now under development, will include notation for representing integer literals in a binary or octal form. The Ada compiler will convert these numbers into decimal form.

#### 2.3.3.4 Type Rational

The Gypsy verification system does not support rational numbers in executable code. They exist in Gypsy as a specification tool. If the compiler discovers a rational number in executable code, it will issue a message and disallow the translation. Note that executable code may include runtime validated specifications, for which executable code must be generated to validate the runtime program state. If rationals appear in non-executable specifications, they appear as normal in the commented specifications.

#### 2.3.3.5 Subrange Types

Subrange types are fully supported in Ada with only minor variation in notation. As previously stated, to duplicate Gypsy type compatibility notions, subrange types must be declared as subtypes. The idea of pre-computability is not discussed in standard Ada documents, but it is reasonable to assume that the Gypsy notion of pre-computability, narrow as it is, could be supported by an Ada implementation.

As long as subrange types are declared with type declarations, they are completely consistent with the initial value and typedescriptor mechanisms utilized for other types in the compiler. A problem arises, however, when symbols are declared to be of a type with the subrange attached to the symbol declaration rather than a type declaration. For example, consider:

```
var fool: integer [1..10];
{ This is opposed to:
      var foo2: smallpos;
  where type smallpos = integer [1..10]; }
```

To determine the typedescriptor of foo2, the compiler needs only recognize that foo2 is of type smallpos and then look up the name of the typedescriptor for smallpos. Fool, however, does not have an independently declared type, hence there is no declared typedescriptor for fool. To remedy this situation, the typedescriptor is generated as an aggregate expression whenever it is needed, for example in an assignment to fool. The GVE may soon be giving these objects of anonymous subrange type special treatment by declaring invisible types. When this treatment is provided, the generation of these aggregates on demand will no longer be necessary.

### 2.3.4 Static Structured Types

Ada has array and record mechanisms which are sufficient as targets for translating Gypsy arrays and records, provided some predefined operations are superseded by new definitions.

#### 2.3.4.1 Arrays

Gypsy arrays map neatly into Ada arrays with a minimum of overhead. The support for an array basetype is entirely declared in a generic package which may be instantiated for each basetype. The units declared in the package include the basetype typedescriptor, the basetype itself, the element selector function, the element alteration function, a valueerror checking function, an aggregate assignment procedure, and an element assignment procedure. The element selector, aggregate assignment, and element assignment procedure are different from those in Ada because of their distinct handling of exceptions. The element alteration clause, as with all alteration clauses, must be provided because no analogous operation is predefined in Ada. Equality and inequality are adequately handled in Ada.

Having these various operations declared as functions and procedures produces quite a notational impact on translated code which utilizes them. Handy and easily recognizable forms such as [i] for indexing and ":=" for assignment must be abandoned in favor of standard function and procedure calling forms. Even these are somewhat unsightly due to the use of qualified global names which are used throughout the compiler. For example, the simple reference to a[1], where a is of type foo, becomes:

```
basetype_package.foo_basetype.select_element
                              (a, foo_descriptor, i)
```

The arguments to the select_element function are the array, its typedescriptor, and the index expression. The global name of the select_element function must include the name basetype_package, from which it was generically declared, the name of the array basetype (here assumed to be foo_basetype), which is the name of the instantiated array basetype package, and the name of the selector function in the package. This is not so much new complexity as it is hidden complexity brought to the surface through translation. Nevertheless, the unfortunate truth is that code which deals heavily with highly structured data types becomes very difficult to read, particularly if the reader is not well-versed in the compiler's operation.

#### 2.3.4.2 Records

The operations defined on Gypsy records are eq, ne, field selection, assignment (both of aggregates and of individual field values), and field alteration. Eq, ne, and field selection are sufficiently handled in Ada. Both varieties of assignment and, of course, alteration, must be provided by the compiler for each record basetype. This involves much more overhead than immediately appears.

Records are the bane of generics in Ada, largely because record field names cannot be used as expressions for parameterization and also because there may be any number of fields in a record. Generic packages could not be useful in generating support code for records, so those functions are generated from templates embedded in the LISP code of the compiler.

Not only may generics not be used, but a function which must deal with an individual record field must be written to operate specifically on that field. Two cases where this complicates the translation are in generation of field assignment code and in generation of alteration clause code. Each field in the record must have its own field assignment procedure and its own field alteration clause. Generating all these functions in the basetype package is somewhat disconcerting when in all likelihood few, if any, of the functions will ever be called.

Other functions generated from templates are the valueerror function for the entire record, the aggregate assignment procedure, and the basetype typedescriptor and initial value declarations.

### 2.3.5 Base Types

Basetypes have been adequately discussed earlier in this section. It is worth noting once again that the compiler is quite basetype oriented, and that typedescriptors provide the most critical guide for type specific behavior in the program at runtime.

The compiler requires basetypes and typedescriptors to have been declared internally in order for most predefined operations to be performed. These declarations are made in conjunction user type declarations. Anonymous types, or types without names, create problems within the compiler. A user may create an anonymous type in one of two ways: 1) by declaring an object of a subrange type, which has already been addressed, and 2) by placing levels of anonymity in type declarations. Consider the type declaration:

```
type seqarray = array ([1..10]) of sequence (10) of integer;
```

The sequence type embedded in the definition is anonymous. An item selected from the array would have no named type.

Ada would not allow such anonymity in type declarations, and we feel that using them is a poor programming practice. It will be disallowed in Gypsy 2.1. Pending the implementation of 2.1, the compiler detects and disallows user specified anonymous types of the embedded variety.

## 2.4 Expressions

### 2.4.1 Name Expressions

Although Ada has same treatment of name expressions as Gypsy, the compiler's abstraction of structured data types and their operations does not allow a normal treatment of structured object names in many places. In those places where normal treatment is sufficient, it is used with glee.

#### 2.4.1.1 Component Selectors

Component selection on structured types is where name expressions demand unusual treatment. In particular, abstraction functions for field selection on structured types other than records must be used rather than direct access. (Direct access would mean normal indexing on arrays. It would mean a clumsy series of linked record accesses on dynamic types.) Also, special treatment is required for element selection on the left hand side of assignments. As explained in section 2.3, since access functions may not appear on a left hand side of assignments and since condition generation is

different, basetype procedures are generated for element assignments. An indexerror on the left hand side is recognized in the appropriate assignment procedure for the object. Name expressions on structured types no longer appear, then, in units directly translated from user code. They are all protected in the abstraction functions for the various basetypes, where they make their only appearances.

### 2.4.2 Value Expressions

The concept of a value expression in Ada is exactly that of Gypsy.

#### 2.4.2.1 Primary Values

The distinction the compiler makes between structured name and value (name) expressions is illustrated by the use of basetype-generated selector functions where value selectors were used in Gypsy. These are clearly illustrated in section 2.3.3.1.

Entry values (primed expressions) are intended for use in specification only. Their use in executable code will disallow the translation. In non-executable specifications, which become comments, they are translated in their original syntax.

Scalar literals are all declared in the basetype package. Therefore, to refer to literal "red" of type "color", it is necessary to use the qualified literal "basetype_package.color_basetype'(red)".

As previously stated, strings are implemented as arrays in Ada and as sequences in Gypsy. The compiler handles them as it would sequences. Translation of a string literal, then is done by coercion of the literal with a predefined function in GYPSY_PACKAGE which takes a string as an argument and returns a CHARACTER_SEQ (a predefined basetype, sequence of character). The string "A string" would translate to the function call:

```
gypsy_package.coerce_string ("A string")
```

Rational values are not translated in executable code. All other primary values are translated directly in to their Ada equivalents.

#### 2.4.2.2 Modified Primary Values

Structure element alterations are performed with functions generated with the basetypes of each structured type. A subsequence selector function is provided in the sequence basetype package.

#### 2.4.2.3 Operators

The treatment of specific operators has been extensively described in section 2.3 on types. Precedence levels are preserved in the Gypsy prefix, and emerge naturally from the compiler.

#### 2.4.2.4 If Expression

No equivalent to the if expression construct exists in the Ada language. Translation of if expressions requires assignment to a temporary, which will serve as a place holder in the statement containing the expression when it is translated immediately afterward. Consider the translation of the statement:

```
a[i] := foo(if a[j] < a[k] then a[j] else a[k] fi);
```

where a is an array of integer values. In translating the statement we would first recognize the presence of an if expression and compute its value, assigning it to the temporary g_a_temp. Then g_a_temp would be substituted for the if expression in the translation of the statement. The generated Ada code would then be:

```
declare
  g_a_temp: integer := 0;
  begin
  if a[j] lt a[k] then g_a_temp := a[j]
```

```
    else g_a_temp := a[k]; end if;
  a[i] := foo(g_a_temp);
end;
```

G_a_temp is declared in the block to be a variable of the basetype of a[j], i.e. integer.

In the case of having more than one if expression embedded in the same Gypsy statement, we would simply have embedded blocks of code. Hence the rather baroque (and useless) Gypsy statement:

```
a[i] := if (if not j in 1..k then j in k..i else j in 1..i)
          then a[j] else a[k];
```

would translate to:

```
declare
  g_a_temp1: integer := 0;
  if (if not j in 1..k then j in k..i else j in 1..i)
    then g_a_temp1 := a[j]
    else g_a_temp1 := a[k];
  a[i] := g_a_temp1;
end;
```

which in turn expands to:

```
declare
  g_a_temp1: integer := 0;
  declare
    g_a_temp2: boolean := false;
    if not j in 1..k
      then g_a_temp2 := j in k..i;
      else g_a_temp2 := j in 1..i;
    if g_a_temp2 then g_a_temp1 := a[j];
                 else g_a_temp1 := a[k];
  end;
  a[i] := g_a_temp1;
end;
```

Gypsy semantics are such that evaluation of Gypsy expressions is free of side effects. This property is required to preserve the semantics of the if expression through this translation. It is sufficient for preservation of semantics, however, only in the absence of conditions and condition handling. Extracting the if expression from the context of a larger expression so that it can be pre-evaluated in an if statement can perturb the order in which conditions can be signalled. Preserving the correct order of evaluation under these circumstances is such an extensive problem that this initial implementation will not attempt to perform it.

### 2.4.3 Pre-computable Expressions

Pre-computability was not defined in Gypsy 2.0, on which the initial implementation of the compiler was based, and the Gypsy 2.0 parser placed very tight constraints on what expressions were regarded as pre-computable. Under these constraints, there is no danger that a pre-computable expression will not be pre-computable upon its translation into Ada.

Gypsy 2.1 liberalizes the previous constraints on pre-computability. Although it stops short of declaring exactly what expressions are pre- computable in a given implementation, it does define the set of expressions which are candidates for pre-computability. The standard documentation on Ada does not address the question of pre-computability, but it is hoped that any good Ada implementation will be able to pre-compute the value of any translated pre-computable Gypsy expression.

## 2.5 Programs

The Ada concept of program is compatible with the Gypsy notion. In particular, those Ada constructs which are targeted by the Gypsy translation are philosophically consistent with the Gypsy constructs, although they will vary in appearance. The effect of translated code on its environment is limited to changing the value of its data objects, raising an exception, and reading and writing files. The external environment specification for each program is provided by the implementor of the compiler for a particular target machine. Ada's concept of procedure, function, and constant is similar to that of Gypsy.

### 2.5.1 Procedures

Ada procedures are fundamentally quite similar to Gypsy procedures. Those aspects which differ are more appropriately discussed in other sections, notably those on textual organization and condition handling and in other subsections of this section on programs. User-defined Gypsy procedures are translated directly into Ada procedures with the same name. A return statement must be inserted at the end of the procedure statement list, since Gypsy procedures normally exit when they execute the last statement.

### 2.5.2 External Environment

Although the Ada concept of the external environment to a procedure is somewhat different from Gypsy, the net effect of translating a procedure into Ada preserves the semantics of the Gypsy procedure. The point of consistency between the environments of Ada and Gypsy procedures is that they are defined by a combination of those objects in the formal parameter lists and those externally visible at the point of declaration of the procedure.

Gypsy procedures have access only to those conditions supplied as actuals to the list of formal condition parameters. Ada procedures have no "exception parameters". Rather, they have access to those exceptions which are declared in any environment which nests the procedure at runtime. Visibility of external constants is defined by the environment of the scope or package which contains the procedure declaration.

While the external environment of an Ada procedure is somewhat different than that of a Gypsy procedure, symbols used in a translated procedure will naturally be only those which were used in the Gypsy procedure plus references to initial value constants and typedescriptors, to which the Ada procedure has access. Ambiguity among symbols in the external environment is eliminated by using global names for all external references. (Ex. The initial value for the basetype of an array type foo declared in scope bar would be basetype_package.foo_bar.initial_value.) Therefore the set of symbols referred to in the Ada procedure represents the same symbols referred to directly or inferred in the Gypsy procedure.

Gypsy "var" parameters to procedures are translated to "in out" parameters in the corresponding Ada procedure. "Const" parameters in Gypsy procedures become "in" parameters to Ada procedures.

### 2.5.3 Functions

The relationship between Gypsy and Ada functions and procedures is the same as between Gypsy and Ada procedures.

Ada has no item analogous to the predefined RESULT of Gypsy functions. Therefore RESULT is declared as a local variable in each translated Ada function. Its type is the type of the function, and its assigned initial value is the initial value for the type. When Gypsy functions terminate normally, the value of RESULT is returned. Ada functions must terminate normally with a RETURN statement which has a parameter, the value to be returned. The value supplied in the translated functions is the locally declared RESULT.

### 2.5.4 Constants

Ada constant declarations are exactly analogous to Gypsy constant declarations, except for the degree to which the concepts of pre-computability vary. This is implementation dependent in both languages, but the rather conservative rules on pre-computability, plus the fact that translated standard Gypsy functions are elaborated before any translated user declarations are given, seems to ensure that any acceptable constant declaration in Gypsy will translate directly into a declaration acceptable to Ada. The Gypsy parser will evaluate constant values, so the question of how to handle exceptions which arise during this evaluation is not of concern.

### 2.5.5 Bodies

Not all of the material which goes into the body of a Gypsy unit is meaningful to the program at runtime. In particular, non-validated specifications have no impact on execution and are converted into Ada comments in translation.

Pending bodies are not wholly acceptable to the compiler, since they reflect that the program translated is incompletely implemented and not ready to be run. If the body of a type or constant is pending, the compiler will abort. If a statement list is pending, the singleton statement list "raise routineerror;" is generated and a message is sent to the tty.

A special transformation is made on all user-declared function and procedure bodies to facilitate condition handling. This is discussed in Section 2.8.

### 2.5.6 Internal Environment

The internal environment of both Gypsy and Ada procedures may include the declaration of local variables, constants, and conditions (exceptions).

The only point of difference involves initial values for variables and constants. All Gypsy objects have initial values, which are either supplied explicitly by the user or assumed to be the default initial value of the type of the object. The internal objects of a Gypsy procedure are created in order of declaration and given their initial values explicitly. This means that the initial value of a symbol may depend on the initial value of any previously declared symbol. In addition, evaluation of the initial value of the local may result in the signalling of a condition. These conditions will propogate directly out of the routine, either through the condition parameter list or as routineerror. They cannot be locally handled by the user, since they are raised in a transitional state of the routine calling mechanism.

The valueerror which can result from the initial value assignment, however, must be treated specially. The initial value assignment in Ada is not an assignment treated consistently with normal assignment statements, since it cannot be replaced in-line with a call to a procedure which can check for valueerror. To perform the valueerror check, the compiler would have to put an explicit assignment statement at the beginning of the procedure for each variable declaration. Direct assignment in the declaration, however, would likely be more efficient, and is at least more closely resembles the Gypsy code.

The compiler attempts to use assignments in declarations whenever possible. It prints the symbol declarations in order. Whenever it finds a declaration where the user has specified an initial value, it attempts to determine if the initial value might generate a valueerror. If it sees there will be no valueerror, it will print the initialization in the declaration and treat succeeding symbols in the same manner. If it cannot determine that valueerror will not occur, it will print a declaration without an initialization and insert a call to the proper assignment procedure at the beginning of the statement list for the procedure. All subsequent declarations must also be made with an explicit call to the assignment procedure in order to preserve the correct order of evaluation. For example, consider the following local declarations in Gypsy (where type small_int = integer [1..10] and there is a formal parameter int_param of type integer):

```
var int1: small_int := 10;
var int2: small_int := intparam;
var int3: small_int := int2 - 1;
```

```
var int4: small_int;
```

These will translate to:

```
int1: small_int := 10;
int2: small_int;
int3: small_int;
int4: small_int := small_int_initial;
gypsy_package.integer_assign (int2, small_int_typedescriptor,
                              intparam);
gypsy_package.integer_assign (int3, small_int_typedescriptor,
                              int2 - 1);
```

Access specifications on internal objects, as with all objects, are ignored for the time being. When data abstraction is implemented in the compiler, access lists will be used to help steer determination of abstract equality, but its effect will be more or less invisible in the translated code.

### 2.5.7 Internal Statements

Of the various statement forms, only the if composition, case composition, loop composition, and begin composition survive intact. The procedure statement is used profusely, but user calls to procedures are embedded in blocks which perform valueerror and aliasing checks. All the others, save the cobegin statement (concurrency is not yet handled), are replaced with calls to procedures provided in the basetype support.

#### 2.5.7.1 Assignment Statement

All assignment statements are converted into calls to the assignment procedure provided for the type of object on the left hand side. The parameters to the assignment procedure are some representation of the name expression on the left hand side, the typedescriptor appropriate for the name expression, and the expression on the right hand side. These procedures first check to see if the value on the right hand side is a legal value for the name expression (using the typedescriptor as a guide). If so, they raise a valueerror exception. Otherwise they make the assignment by invoking Ada assignment procedures.

The name expression on the left hand side may have a selector list with as many selectors as it takes to descend through a data structure to whichever component is to receive the expression on the right hand side. This creates a serious problem in Ada translation, since assignment statements for structured objects are generated as generic procedures, and these procedures must have the entire variable structure as a parameter. The only such procedures generated for structured types are the structure assignment and the component assignment (one level deep).

The only reasonable solution, then, is to decompose the assignment into assignments to temporaries one level at a time. For example:

```
type foo = sequence of integer;
type bar = sequence of foo;
...
var a: bar;
a[i][j] := 10;
```

would translate into:

```
var a: bar;
declare
  var g_a_temp: foo;
  foo_basetype_assign (g_a_temp, a[i]);
  foo_basetype_element_assign (g_a_temp, j, 10);
  bar_basetype_element_assign (a, i, g_a_temp);
end;
```

It is not difficult to see that this mechanism is clumsy and grossly inefficient at best, and becomes more so as the levels of descent into a structure on the left hand side increase. With structured types implemented as they are, however, it is the only way in which the semantics of structured component assignments may be maintained.

#### 2.5.7.2 Input and Output

Input and output of Gypsy programs, as currently supported, is performed entirely through buffers. Those buffers which are parameters to the main program are designated as I/O buffers rather than normal internally used buffers. The send and receive operations on these buffers are the I/O operations print and read.

Ada input and output is performed through a predefined Ada package SEQUENTIAL_IO. This package must be instantiated for each different type of object which will be involved. Thus, if both characters and integers will be output, two package instantiations must occur, with each defining a set of input and output functions and file types which may be used with the appropriate data objects. The file types are IN_FILE, OUT_FILE, and INOUT_FILE for each type of object. "Internal" files are then declared to be of these file types in the same manner as any other declaration. The package instantiations and internal file declarations occur at the end of basetype_package. For example:

```
-- Here we instantiate the I/O package for character types

package character_io is new sequential_io
                            (element_type => character);

-- This is a declaration of an "internal" file for output of
-- characters

char_file: character_io.inout_file;
```

To associate devices, or external files, with "internal" files, the CREATE, OPEN, and CLOSE procedures are invoked. These associations are made in the supermain routine which is the calling point for the compiled program. For example, the following statement will open the file whose device name is the string value of the variable FILENAME. This could, for instance, be "<cmp.akers>inchars.history".

```
basetype_package.character_io.open
  (file => basetype_package.char_file, name => FILENAME);
```

An operation to write to the file would look like:

```
character_io.write (file => basetype_package.char_file,
                    item => 'a');
```

A further description of the manner in which input\output objects are treated is given in the sections on the main program and implementation prelude.

Since Gypsy buffers may be passed around as parameters, the routines which call send and receive have no way of statically knowing whether they are performing input/output or internal message sending. Since the Ada input/output mechanism is different from the mechanism employed to perform buffer operations, some distinction will be needed at runtime in the Ada translation to determine what kind of operation to perform.

To this end an extra integer field will be included in the Ada image of Gypsy buffer objects. The field, named io_flag, will have the value 0 if the buffer is for internal use, and a positive value which associates it with one of the internal files if it is an input/output buffer. The value corresponds to the ordinal position of the buffer in the parameter list to the user's designated main procedure. A send or receive statement is translated into a case statement which keys on the io_flag field of the buffer. If the value is 0, a normal send to an internal buffer is performed. If it is non-zero, a write or read for the appropriate internal file is generated. Here is the Ada image of the Gypsy statement "send 'a to buf;":

```
if gypsy_package.character_valueerror
            (buftype_descriptor.b_elem_type, 'a')
        then raise valueerror;
else case buf.io_flag is
      when 0 => basetype_package.buftype_test.send
                  ('a', buftype_descriptor, buf);
      when 1 => basetype_package.character_io.print
                  (basetype_package.char_file, 'a');
      when others => raise caseerror;
     end case;
end if;
```

#### 2.5.7.3 If Composition

The Gypsy if composition statement form is exactly the same as the one in Ada, excepting minor points of syntax, and is translated directly.

#### 2.5.7.4 Case Composition

The Ada case composition statement is like that of Gypsy, except that the when others clause, corresponding to the else clause of the Gypsy case statement, is required. If there is no else clause in a Gypsy case composition and none of the case arms match the value of the label expression, a caseerror is signalled at runtime. In order to duplicate this behavior, an absent else clause is translated into an Ada when others clause which explicitly raises the caseerror exception.

#### 2.5.7.5 Loop Composition

The loop statement in Ada is somewhat more general than the Gypsy loop composition, but it includes a syntactic and semantic form which exactly matches the Gypsy construct. The translation is direct.

### 2.5.8 Procedure and Function Calls

The Ada procedure and function calling mechanisms are fundamentally similar to Gypsy, although they are somewhat more flexible. To correctly handle procedure and function calls, however, the compiler must ensure that conditions which may be signalled from the call are generated and handled according to Gypsy rules. The solution to this problem is somewhat different for functions than for procedures.

Aside from conditions which propogate out of the called function, the only condition which may be signalled from a Gypsy function call is the valueerror which may be raised when an actual parameter violates the type constraints of its formal. In Ada this would result in a constraint_error (or in the case of translated dynamic types would likely go undetected). In either case this must be prevented. Since function calls may be embedded in arbitrarily complex expressions, it would be prohibitively complicated to check for valueerror at the call site. Neither can the checks be made in the called function since Ada would have by that time detected the violation through its own calling mechanism.

To resolve this difficulty, each user defined function is declared with an interface function. This function is called rather than the original, or parent, function. Its formal parameters are the same as those of the parent, except that their types are the basetypes of the corresponding formals in the parent. The type of the function is the basetype of the type of the parent. The function checks each of its parameters for valueerror against the type of the corresponding parent formal, and raises the valueerror exception if it finds a violation. This exception will propagate back to the call site, producing the same behavior as Gypsy specifies. If there are no violations, the parent function is called, and its value returned as the value of the interface function. For example, consider the following function declaration:

```
function add1 (i: int): int =
 begin
  result := i + 1;
```

```
end;
```

The function add1 is renamed to g_a_add1 and a new function add1 is defined as follows:

```
function add1 (i : in integer) return gypsy_package.int is
 result : gypsy_package.int := 0;
 begin
  if gypsy_package.integer_valueerror
              (gypsy_package.int_typedescriptor, i)
         then raise valueerror;
    end if;
  result := g_a_add1 (i);
  return result;
 end add1;

function g_a_add1 (i : in gypsy_package.int)
                   return gypsy_package.int is
 result : gypsy_package.int := 0;
 begin
  gypsy_package.integer_assign
       (result, gypsy_package.integer_plus (i, 1),
                gypsy_package.int_typedescriptor);
  return result;
 end g_a_add1;
```

If the types of the parameters to the user function are all basetypes, the interface function serves no purpose, since valueerror may not occur. In this case the interface function is not declared.

Valueerror checking must be provided on procedure calls also. Since the procedure call is a free-standing statement (as opposed to the function call, which may be embedded in a complex expression), the checks may be done at the call site without complication. The called procedure need not be affected by the translation of the calling mechanism. For example, consider the following procedure defined in scope s (as above):

```
procedure amin (var i: small_int; a: small_int_array) = ...
```

A call to amin of the form "amin (i_actual, a_actual);" would need the structure:

```
begin
 if gypsy_package.integer_valueerror (small_int_descriptor,
                                      i_actual)
   then raise valueerror; end if;
 if basetype_package.small_int_array_s.valueerror_occurs
       (small_int_array_descriptor, a_actual)
   then raise valueerror; end if;
 amin (i_actual, a_actual);
end;
```

Checking for aliasing on var parameters, discussed later, may also be embedded in this structure.

There are two problems with this scheme of procedure calling. One is that if an actual parameter is an expression which must be evaluated to produce a value, the evaluation will be done twice: once in the call to the valueerror function and once in the call to the procedure itself. The other problem occurs where the actuals corresponding to var formals are name expressions which index into a structured item (Ex. a[i] or rec.field1). In such a case, the Ada representation of the actual is a function call (the function which performs indexing on objects of the basetype of the root of the actual). This violates the constraint that variable names must be used as actuals where the formal is an in or in out parameter.

In-line emulation of the value-result parameter passing scheme will solve both of these problems simultaneously. In this scheme, the value of the actual is calculated and assigned to a temporary of the type of the formal. The temporary is used as the actual parameter in the procedure call itself, and its value upon return is re-assigned to the variable from which it was created. Note that the expression is evaluated only when it is assigned to the temporary. It is checked for valueerror in the

assignment, since the assignment will be performed by the assignment procedure for the basetype of the formal with the typedescriptor of the formal. Any conditions arising out of evaluation will arise in the proper order with respect to the rest of the computation. The temporary has a name, which satisfies the constraint for actuals of var parameters. The reassignment to the original actual will be performed with the typedescriptor for the original actual, so valueerror will be checked correctly and at the appropriate time. Incidentally, the checks for aliaserror may be inserted cleanly between the assignments to temporaries and the call to the procedure, so that any conditions arising from that check will appear at the appropriate time.

Consider this Gypsy example, particularly the call to proc:

```
scope bim = begin
  type foo = array ([1..10]) of integer;
  procedure proc (var i1: int; i2: int) = pending;
  procedure caller (var arr: foo; i,j: integer) =
   begin
    proc (arr[i], arr[j]);
   end;
end;
```

The call "proc (arr[i], arr[j])" becomes the block statement:

```
declare
 g_a_temp1, g_a_temp2 : gypsy_package.int;
 begin
  gypsy_package.integer_assign (g_a_temp1,
     basetype_package.foo_bim.select_component (arr, i),
     gypsy_package.int_typedescriptor);
  gypsy_package.integer_assign (g_a_temp2,
     basetype_package.foo_bim.select_component (arr, j),
     gypsy_package.int_typedescriptor);
  if i = j then raise aliaserror; end if;
  proc (g_a_temp1, g_a_temp2);
  basetype_package.foo_bim.element_assign
      (arr, foo_descriptor, i, g_a_temp1);
 end;
```

#### 2.5.8.1 Actual Parameters

The treatment of actual parameters is embedded in the previous discussion. The runtime vararror check which was required in Gypsy 2.0 is no longer necessary in Gypsy 2.1, due to the disallowance of local declarations whose types are restricted by the values of formal parameters.

#### 2.5.8.2 Type Consistency

Type consistency has been thoroughly discussed above.

#### 2.5.8.3 Aliasing

Some alias checking, specifically the check to see if a candidate pair of actual parameters are exactly the same name expression, may be performed by the parser. The rest of the check must be performed at runtime. The compiler must check for pairs of type-compatible formal parameters where at least one is a variable parameter and the roots of both name expressions are the same. In this case, the indexes into each expression must be checked at runtime for equality. If they are equal down to the finest index of the shallowest name expression (a.b[1] is shallower than a.b[1].c.), aliasing exists and aliaserror must be signalled. Consider the procedure header and call below:

```
type int_array = array ([1..10]) of integer;
type array1 = array ([1..10]) of int_array;
type seq1 = sequence of array1;
procedure proc (var i,j: integer) =  ...
```

```
procedure foo (var s1, s2: seq1) = begin
 var i1, i2, i3, i4, i5, i6: integer;
 ...
 proc (s1[i1,i2,i3], s2[i4,i5,i6]);
```

For the sake of clarity in the example, let us consider that the valueerror checks have been optimized away and that for the moment we need not be concerned with the conversion of the translated actual parameter expressions into name expressions. The aliaserror check will be included in the procedure calling mechanism in the following manner:

```
if (i1 eq i4) and (i2 eq i5) and (i3 eq i6)
  then raise aliaserror
  else proc (s1[i1,i2,i3],s2[i4,i5,i6]);
```

This statement will be inserted in the procedure call block immediately preceding the call itself. (See also the last example in Section 2.5.8.)

#### 2.5.8.4 Transfer of Control

The code to transfer control through a procedure call is created so as to preserve the order of operations described in the Gypsy manual. In particular, the aliaserror check will be made after the valueerror check and may occur either before or after the creation of name expressions to use as actual parameters.

### 2.5.9 Getting Started

#### 2.5.9.1 Verification Environment

The generation of Ada code from Gypsy code is directed entirely from within the Gypsy Verification Environment. The exec command in the environment which orders a translation is the translate command. The command is menu driven by the exec grammar.

The translate command initiates an interactive dialogue with the user to acquire necessary information. An annotated dialogue may be found in Appendix E.

#### 2.5.9.2 Main Program

When the user invokes the compiler, he supplies the name of an implementation prelude to serve as the environment for the execution of his program and a call to a procedure which is interpreted to be the main program. The main program specified must be a Gypsy procedure. As currently supported, only buffers are allowed to be variable parameters to the main procedure. These buffers are treated specially throughout the Gypsy program. (See the section on input and output.) The types of all parameters to the main program must have their basetype defined in the given implementation prelude. In the user's call to main, the actual parameters are names chosen from the "variables" in the chosen environment. These variables must be type-compatible with the formals in the user's main procedure. Note that in the prelude, the range restriction constants are "pending", which means in this case that they will be supplied by the compiler to match those of the formals in the user's main procedure.

The main procedure is transformed into three separate procedures which are all placed in the same package as the user's main. If the user's main were named, for instance, TOP_PROC, the compiler would produce procedures named G_A_SUPERMAIN, G_A_TOP_PROC_ENTRY, and TOP_PROC. TOP_PROC itself is not treated specially in the translation. Its header, however, is used as the basis for the generation of an intermediate procedure, G_A_TOP_PROC_ENTRY, which sets the io_flag fields of the buffer parameters and does valueerror checking before calling TOP_PROC. G_A_SUPERMAIN is the outermost procedure, and it serves to link the program with its execution environment and determine the system-level devices which are being used by the program.

Consider the following procedure which is designated as the main program.

```
procedure top_proc (var b: buftype) = begin
bufproc (b);
end;
```

This procedure itself is translated in a straightforward manner, as follows.

```
procedure top_proc (char_file : in out buftype) is
 begin
  declare
   g_a_temp1 : buftype;
   begin
    basetype_package.buftype_test.assign
       (g_a_temp1, char_file, buftype_descriptor);
    bufproc (g_a_temp1);
    basetype_package.buftype_test.assign
       (char_file, g_a_temp1, buftype_descriptor);
   end;
  return;
 end top_proc;
```

This version of top_proc will behave exactly like the originally defined function for all internal calls.

The compiler must take several steps in the main procedure to set up the environment of the program. These steps are embodied in g_a_top_proc_entry. To set up correct input and output interfaces, the compiler must examine the formal parameter list of the main procedure. Each buffer parameter is treated as an input/output file but is type consistent with normal buffers. So that internal code doing sends and receives may differentiate between regular buffers and input/output buffers, an integer field in the buffer record structure indicates the buffer status. If the field is set to 0, the default, the buffer is a regular internal buffer. If it is non-zero, the value represents the ordinal value of the buffer in the parameter list to main, i.e., if set to 1, it is the first parameter to main, if set to 3, it is the third. The entry procedure first assigns these values to the buffer io_flag fields, then simply calls the user's main procedure. The above declared main procedure will produce the following entry procedure, which will be declared just after the main procedure.

```
procedure g_a_top_proc_entry (char_file : in out buftype) is
 begin
  char_file.io_flag := 1;
  declare
    g_a_temp2 : buftype;
    begin
     basetype_package.buftype_test.assign
       (g_a_temp2, char_file, buftype_descriptor);
     top_proc (g_a_temp2);
     basetype_package.buftype_test.assign
       (char_file, g_a_temp2, buftype_descriptor);
    end;
   return;
  end g_a_top_proc_entry;
```

The procedure which serves as the actual call point to the program is g_a_supermain. It forms the link between the execution environment and the translated program, declaring the buffer variables which are formals in the user's main procedure and querying the user for device names which are then associated with the internal files declared in basetype_package. G_a_supermain opens these files, calls the entry procedure, and closes the files on return. If a file corresponds to an output-restricted buffer, the file is opened with a CREATE statement; otherwise it is assumed that the file exists, and an OPEN statement is used. In generating g_a_supermain, the compiler uses the names of the formal parameters of the user's main as names for the actuals which it declares. It has access to the internal representation of the user's invocation to main, so that it can use the user's actuals as keys for querying for device names. TTYIN, TTYOUT, and TTY are treated as special case actuals for which a device name is not needed. Here is the supermain procedure generated for the top_proc example.

```
procedure g_a_supermain is
```

```
filename : gypsy_package.string;
char_file : buftype := buftype_initial;
begin
  begin
    tty_io.put (" FILE FOR IN_CHAR_FILE1 ? ");
-- IN_CHAR_FILE1 is an object from the implementation prelude
    tty_io.beep;
    tty_io.get (filename);
    basetype_package.character_io.open
      (basetype_package.char_file, filename);
  end;
  g_a_top_proc_entry (char_file);
  basetype_package.character_io.close
      (basetype_package.char_file);
  return;
end g_a_supermain;
```

#### 2.5.9.3 Implementation Prelude

The implementation prelude is a collection of type and constant declarations, plus an environment unit which declares as variables all of the Gypsy objects which may be used as actual parameters in the user's call to his main program. Typically, there are several objects of a given type, so that more than one may be used in the program. Each must be unique in the user's actual parameter list. The user's call is checked for semantic correctness by a call to the Gypsy parser which utilizes special semantic rules. An example of an implementation prelude may be found in Appendix C. Its utilization by the compiler is described in the preceding section.

## 2.6 Operational Specifications

Most Gypsy specifications have no effect on the runtime performance of the program. Where this is the case, the compiler transforms them into comments.

There are a several Gypsy forms which are not effectively translatable into executable Ada code. Most of these are in the language as specification tools. Some of them, such as quantified expressions and rational numbers, have been noted earlier. If the compiler encounters one of these forms when transforming a specification expression into a comment, it will attempt to regenerate it in its original form. If it finds a non-translatable feature in a specification containing a validation directive, it will emit a message and abort the translation.

Gypsy allows functions which contain only external specifications to be written. When such a function is encountered in translation, it is regenerated as a comment. Runtime validation on such a function is meaningless, since the function is regarded to have a pending body and thus cannot execute. Similarly, it is meaningless for a specifications-only function to be called from a specification which is being runtime validated. In either case, the compiler will abort the translation.

### 2.6.1 Specification Expressions

While most Gypsy specifications have no runtime impact and may become comments, runtime validated specifications must be treated in executable code generated by the compiler. A validated specification is composed of a boolean expression indicating a validation criterion and a condition to be signalled if the validation fails. Translating such specification involves generating code to evaluate the boolean expression, signal the condition if the result is false, and proceed quietly if it was true. In Gypsy 2.1, specification expressions have only one part, the boolean expression, so that the transportation of the expression into executable code will be monolithic. Consider the validated specification, where a is of type arr = array ([1..10]) of integer;

```
(array_summation (a) lt 1000) otherwise summation_error
```

This specification will translate into a statement, which will be appropriately placed in the program text. The statement will be:

```
if not (array_summation (a) lt 1000)
  then raise summation_error;
```

#### 2.6.1.1 Entry Values

Entry values, or primed variable object names, are intended for specification use only, and thus will result in an aborted translation if detected in a validated specification (or in executable code). Otherwise, they are printed normally.

#### 2.6.1.2 Value Alterations

For each basetype of a record or array, the compiler declares a function for performing a value alteration of a given field. For arrays, the function takes the array, the index of the replaced value, and the new value as parameters and returns an array of the same basetype with the modified value. For record basetypes, one function is declared (unfortunately, but necessarily) for each field. Each of these functions takes the record and the new value as parameters and returns a record of the same basetype with the modified value.

Compositions of alterations may be expressed in Gypsy which perform multiple alterations on a structure. These may be expressed using either an explicit list of alterations or an each clause which will iterate over a specified index, replacing an item with each iteration. For an example of the former case, consider the Gypsy expression:

```
a with ([1] := 10; [2] := a[1])
```

where a is of type arr = array ([1..10]) of integer. This will produce a recursive function call to the alteration function for the basetype of a (for brevity, call it alter_fn and call the array selector function select_fn) which will appear as:

```
alter_fn (alter_fn (a, 1, 10), 1, select_fn (a, 1))
```

Each clauses are not handled in this manner. Since they are fundamentally looping constructs, an alteration clause containing an each clause will be replaced in the statement in which it appears by a temporary whose value will be computed immediately prior to the statement. Consider the statement below, where a and b are of type arr and the array assignment procedures are for the moment optimized back to ":=".

```
b := a with (each i: integer[1..10], [i] := i);
```

This statement could be replaced with the block:

```
declare
 g_a_temp: arr_basetype;
 g_a_temp := a;
 for i in 1 .. 10
   g_a_temp[i] := i;
  end loop;
 a := g_a_temp;
end;
```

Due to the flexibility of the each clause, it is not possible to implement a generic function which could hide the details of this process.

Alterations with each clauses, like the if expression, require the normal order of evaluation to be rearranged in a manner difficult to retrace. Moreover, the internal representation of alteration clauses in the GVE is being reviewed. For these two reasons, the implementation of each clauses has been delayed. It is possible that, when full condition handling is implemented, they, like the if-expression, will no longer be translatable.

#### 2.6.1.3 Quantified Expressions

Quantified expressions are intended for specification only and are not translatable into executable code.

### 2.6.2 External Specifications

Of all external specifications, only entry specifications may be runtime validated. If an external specification is to be validated, the resulting statement will become the first executable statement of the unit.

### 2.6.3 Internal Program Specifications

Of the two types of internal program specifications (keep and assert), only assert specifications may be runtime validated. The statement produced by translating a validated assert will replace the assert statement in the program text, with the assertion itself printed as a comment.

### 2.6.4 Lemma Specifications

Lemmas have no runtime impact on the program, and are thus translated as comments.

## 2.7 Textual Organization

The treatment of scopes, name declarations, and name resolution will change somewhat in the transition from Gypsy 2.0 to Gypsy 2.1. The compiler currently implements the Gypsy 2.0 mechanism. A description of its treatment follows immediately. After that will come a description of how the Gypsy 2.1 mechanism will be implemented in the compiler.

Conceptually, Gypsy scopes and Ada packages share much common ground. Both are schemes for modularization of code. Both are mechanisms for restricting access among program regions and for isolating the name spaces of sectors of the program. On account of this strong commonality, preserving the scope structure of a Gypsy program by translating scopes into packages is worthwhile. There are, however, important differences between the two constructs which need to be analyzed and dealt with carefully.

First, there are significant differences in the ways which scopes and packages interact with each other. Gypsy scopes are collections of declarations of procedures, functions, lemmas, types, and constants, all of which may be referred to anywhere in the scope. Also included are name declarations, which are the only mechanism for accessing units from another scope. A name declaration brings the named units into the name space of the importing scope. If the <unit id> alone is used as a named-in item, the unit is referenced in by the same name in the imported scope as it had in the exporting scope. If the <identifier> = <unit id> form is used, the unit is renamed to <identifier> in the importing scope. In either case, the name declaration must not introduce an identifier of the same name as one elsewhere declared globally in the scope. Only units may be named into a scope; there is no mechanism for bringing a whole scope into the name space of another scope other than naming in each of its units individually.

This presents a stark contrast to Ada packages, which have no facility for selectively bringing single units from another package into the package name space. If a package needs access to units from another package, a WITH clause in the package header will provide this access for all units in the included package. While a WITH clause provides access, it does not bring the units into the local name space of the importing package. To reference a unit foo from package bar in package baz (where baz was preceded by a "WITH bar;"), the form "bar.foo" is required. To actually bring the units of bar into the local name space of baz, a USE clause must be included in baz's header. The units of bar may then be referred to without the package name qualification, except where the local name is ambiguous.

Translating the effect of name declarations in Gypsy scopes to Ada packages is easily accomplished. The USED attribute of scopes in Gypsy prefix gives a list of all scopes which are

referenced within a given scope. Each of these scopes may be included in the WITH clause of the package. Although this gives the package access to units which it could not access in the Gypsy scope (i.e. those units in the exporting scope which are not explicitly named in), no references to these units appear in the code. Their inclusion, then, is safe.

Although there is a renaming declaration in Ada, with which any accessible unit may be renamed, it is not advisable to attempt renaming of those units which are renamed across scope boundaries in Gypsy. The problem occurs when attempting to rename functions and procedures. An Ada renaming declaration is of the form:

```
RENAMING DECLARATION  ::=
   IDENTIFIER : TYPE_MARK renames NAME;
 | IDENTIFIER : exception renames NAME;
 | package IDENTIFIER renames NAME;
 | task IDENTIFIER rename NAME;
 | SUBPROGRAM_SPECIFICATION IDENTIFIER renames NAME;
```

A Gypsy name declaration requires only the name of the exported routine. That the routine's parameter list may include references to type names which were imported from yet another scope is of no consequence. An Ada subprogram specification, however, includes the entire header for the routine, which encompasses the parameter list and, if the unit is a function, the type of the value returned. If the parameter list or function type refers to a type which is not in the same scope as the function, and that type is not in a scope which is among those being named in via a WITH clause, then an undefined symbol is generated when the type name is printed with the unit header. This symptom could be treated by including the name of the scope in which the type resides in the WITH clause, but in so doing, the scoping of the translated Ada code begins to lose its resemblance to that of the original Gypsy program. This would also introduce a greater likelihood of circular references among packages. While the renaming may have been necessary to eliminate name conflicts in the Gypsy scope, it accomplishes very little in the Ada version, where name conflicts may be resolved simply by using the global name of the unit (with its home package appended).

The Ada USE clause will bring the units of an imported package into the local name space of the importing package, so that global references are not necessary (i.e., so the package name need not be appended to the unit name). In the absence of renaming, the same name conflicts avoided in the Gypsy program could recur in the Ada version if the imported package were USED. A decision would then need to be made as to when the ambiguity exists, so that it could be eliminated by using the global reference. This is more overhead than is merited by the rewards. The USE clause and renaming declarations in the Ada text are not attempted. All references to imported units are with global names.

The Ada requirement that all units must be declared before they can be referenced has caused a certain amount of grief. While it is possible to use the division of a package into a header and a body to declare some items before they are fully defined (which allows for recursive function calls and linked record structures), packages at the top level may not in any way refer to each other in a cyclic manner. One of the package headers has to come first, and it must give the name of the other package, which has not been declared, in its WITH clause. In Gypsy, such cyclic references among scopes are allowed. For example:

```
scope a = begin
name b1 from b;
type a1 = integer [1..b1];
end;
scope b = begin
name a1 from a;
const b1 = 10;
end;
```

If the WITH clause were mutually applied between packages a and b, the undeclared symbol problem would arise. This problem could be corrected by 1) lumping all user defined units into a single package and taking pains to eliminate name conflicts, or possibly by 2) restructuring the user defined units into a hierarchical package configuration so that cyclic package accessing is not necessary. Both of these solutions violate the desired goal of maintaining in the translated Ada code a good semblance of the original Gypsy program. An unfortunate, yet preferable solution is to restrict the Gypsy code

given to the compiler so that cyclic references may not be made among its scopes. With this imposed hierarchy, it is then possible to order the packages so that no forward references to other packages are made. Violations of this restriction may be detected by the same operation which determines the ordering.

The units within a package must be ordered so that the problem of reference to an undeclared symbol does not occur. At this level, the problem appears to be more complex, but in fact the solution is easier. The kinds of units involved are types, constants, functions, procedures, and lemmas. Lemmas are not translated, so we need not address them. For the purposes of name scopes, functions and procedures are identical. Constants declarations may refer to a type name, and they may also refer to other constants. Naturally, it makes no sense for constant declarations to refer to each other in a circular manner. Constant declarations may not refer to user functions, even for initial value specification, since user function calls are not regarded in the verification system to be parse time computable. Types may refer to constants, as in a range or size restriction. They may also refer to other types, but not to themselves and not to other types so that a recursive types are defined. As with constants, type declarations may not, in the current Gypsy implementation, include function calls.

The resolution of forward references, then, can make use of these various relations, and of the fact that in Ada packages, a unit may have its header declared in the package header, but its body need be given only in the package body. Functions and procedures, since they may not be referenced in either type or constant declarations, may have their declarations saved for last. Since their headers will refer only to types and constants, they may be declared in any order at the end of the package header. In their bodies they may, of course, call other functions and procedures, but by the time these references appear in the package body, the headers of all functions have already appeared. This resolves the ordering problem for functions and procedures.

Types and constants may refer to each other in a random order, but they may not refer to each other in a cyclic manner. This reduces the problem, then, to ordering them according to their forward references, which may be accomplished in the same manner as with scopes, i.e. by using the USED attribute of each unit. The header and body of both constants and types may then be fully declared in the package header. This is in fact necessary, since functions and procedures may refer to their composition, which requires them to have been fully declared already.

Most of this treatment will carry over to the Gypsy 2.1 implementation. The transition will involve adding several new mechanisms to accommodate the additional complexity of the revised approach to scopes, name declarations, and name resolution.

### 2.7.1 Scopes

The presence of the EXPORT and IMPORT in the headers of scopes will likely have no impact on the form in which scopes are presented to the compiler. Whereas in the 2.0 implementation the compiler gave a package access (some of which was not needed and was never exercised) to more units than the Gypsy scope had, now the access mechanism provided in translated code will be more like that in the Gypsy program.

### 2.7.2 NAME Declaration

The possibility of aliasing chains via name declarations in Gypsy 2.1 complicates the compiler's approach to name declarations considerably. A unit which may be referenced by an alias may not reside in a scope to which the referencing scope has access. The normal compiler technique of referencing non-local names is to use the global name, but this may only be done where the referencing package has access to the package containing the definition of the referenced unit. Giving the referencing package access to all packages which contain the root definitions of aliased units would not only be complicated, but it would greatly increase the possibility of circular references among packages, an already strong constraint.

To deal with this problem, the compiler will generate a unit declaration wherever a name declaration occurs. The declaration will carry the same header as the unit being aliased, except that the aliasing name will be the name of the unit. The body of the new unit will be a simple reference to the original unit. Consider the following examples:

```
scope a =
 uses b;
 for c;
 name foo from b;
 name bim = bar from b;
end;
scope b =
 for a;
 type foo = integer [1..10];
 procedure bar (var i: integer) = ...
end;
```

The name declarations in package a will become unit declarations as follows:

```
subtype foo is b.foo;
procedure bim (in out i: integer) is
  begin
    b.bar (i);
  end bar;
```

With these dummy units declared in package a, any references to foo or bim in package c will have indirect access through package a to the original unit definitions in package b.

### 2.7.3 Unit Name Reference Resolution

Although the symbol table design for Gypsy 2.1 has not been done, it will need to include some kind of table to map local unit references to global references. This table will be usable to the compiler, which will again place the global references themselves in the code where they are used.

## 2.8 Condition Handling

Translating Gypsy conditions into Ada exceptions would appear straightforward, since the basic mechanisms are conceptually very similar. There are, however, important differences which make the translation non-trivial.

### 2.8.1 External Conditions

In Gypsy the external conditions to a user routine are only those which are named in the formal condition parameter list, plus routineerror and spaceerror. Ada routines do not have any mechanism analogous to the condition parameter. External exceptions include any which have been declared in the external environment of the routine. Multiple declarations of the same exception in any environment are redundant and of no significance.

Gypsy's handling of conditions across routine boundaries can map fairly neatly into Ada, provided that conditions are not renamed in the actual/formal mapping in the Gypsy program. This constraint is enforced by the compiler; a violation will result in an aborted translation. Also, the set of predefined Gypsy conditions are treated as reserved words in the compiler. (This is no problem since they are reserved words in Gypsy.) This will protect the condition handling mechanism for predefined Gypsy functions and procedures translated into Ada.

### 2.8.2 Internal Conditions

Internal conditions may be declared in Ada in much the same way as in Gypsy, i.e. in the local symbol declaration segment of the routine. A Gypsy condition declaration "cond fooerror;" will translate to Ada as "fooerror: exception;".

### 2.8.3 Signalling Conditions

Ada exceptions may be raised in the same manner as Gypsy conditions, i.e. with a RAISE statement or in a procedure call. The Gypsy caseerror, which may arise out of a case statement, is raised explicitly in translated case statements. Gypsy spaceerror is totally implementation-dependent, hence its treatment will be superseded by Ada's storage_error.

#### 2.8.3.1 Signal Statement

The Gypsy signal statement translates directly into the Ada raise statement. The only Gypsy conditions which may be mentioned are those locally declared in the routine, either as a condition parameter or with a local symbol declaration. The exception mentioned in an Ada raise statement may be any one which is visible in the environment of the call. The mechanism by which these two concepts are made consistent is described later.

#### 2.8.3.2 Actual Condition Parameters

The constraint that formal/actual pairings must use identical condition names allows the use of actual condition parameters, or any analogous mechanism, to be dropped from the call site.

### 2.8.4 Handling Conditions

Ada condition handlers are very much like Gypsy, varying only in syntax. The primary difference is the way in which conditions propagate when they are not handled locally. If a Gypsy condition is not handled in a particular routine, it will either propagate through the condition parameter list as a signal to the actual condition at the call site, or if the condition is not in the external environment, it will be transformed to a signal to routineerror at the call site. Ada exceptions, on the other hand, propagate across the routine boundary to the call site intact, in effect always resulting in the same exception emerging from the routine call.

To map the Gypsy mechanism into Ada requires an exception handler to be wrapped around the body of each user routine. In it each formal condition parameter is explicitly trapped and re-signalled. The when others clause of the handler catches all other exceptions and transforms them to routineerror. Thus each exception which may escape the internal environment of the routine is explicitly raised by this outer handler. All internally raised exceptions are caught by the handler and treated according to Gypsy semantics. No special treatment is required at the call site. Consider this example:

```
procedure foo (var i: integer) unless (cond1, cond5) =
 begin
  var j: int := 0;
  cond cond2;
  cond cond3;
  cond cond4;
  if i lt j then signal cond1;
   elif i lt 100 then signal cond2;
   elif i lt 1000 then signal cond3;
   else signal cond4; end;
  when is cond2: i := 0;
       is cond3: raise cond5;
 end;
```

This procedure will translate to:

```
procedure foo (in out i: integer) is
  j: gypsy_package.int;
  cond2, cond3, cond4: exception;
  begin
    gypsy_package.int_assign
      (j, gypsy_package.int_descriptor, 0);
    begin
```

```
      if i le j then raise cond1;
        elsif i lt 100 then raise cond2;
        elsif i lt 100 then raise cond3;
        else raise cond4; end if;
      exception
        when is cond2 => i := 0;
        when is cond3 => raise cond5;
     end;
    exception
      when is cond1: raise cond1;
      when is cond5: raise cond5;
      when others => raise routineerror;
end;
```

In this situation signal to cond1 would propagate out of foo into the calling environment through the outer handler. The signal to cond2 would be caught by the inner handler, as in the original procedure, and would never emerge from foo. The signal to cond3 would be caught by the inner handler and re-signalled as cond5, which in turn would be caught by the outer handler and propagated as cond5. The signal to cond4 would be caught by the when others clause of the outer handler and propagated as routineerror. In each case the Gypsy behavior is recreated. The call site may drop its actual parameters and will otherwise remain unaffected.

Note that this example illustrates the concept of an "extended body" which is implied by Gypsy semantics. Local initializations occur as the first statements of the routine (except in the case where there is a runtime validated entry specification, when the statement performing the validation immediately precedes variable initializations). Then follows the block which is the image of the user's routine body, including his outermost exception handler. The special exception handler generated by the compiler is wrapped around all of these statements.

Note that if the compiler allowed renaming of exceptions through the formal/actual mapping, special treatment would be required at all procedure call sites. Exceptions would need to be treated through some other mechanism, for instance by including an extra integer var parameter which would return 0 normally and, if an exception were propagating out of the called routine, the ordinal number in the list of n formal condition parameters of the exception being raised (n+1 for routineerror). Then each procedure call would need to be followed by a case statement which would transform each non-zero value of the extra parameter into the appropriate exception at the call site (or be a no-op for a value of 0). The situation would be even more complicated for function calls, which could be embedded arbitrarily in expressions. This complexity and our perception that formal/actual renaming is not a critical feature (we have not observed its use in any real Gypsy program), are the reasons justifying the compiler's rejection of that feature.

The correct handling of Gypsy standard conditions is potentially complex and perilous. Difficulties can arise primarily because there are a multitude of standard conditions in Gypsy, and only five standard exceptions in Ada. In many cases, a large number of Gypsy conditions collapse functionally into the same Ada exception. For example, the Gypsy conditions adderror, subtracterror, multiplyerror, divideerror, minuserror, powererror, and powerindeterminate all are subsumed into the Ada numeric_error. The problem of this fine granularity is that the user may write individual handlers for each of the Gypsy conditions. To preserve the semantics of these handlers, each arithmetic operation subject to a condition handler would need to be isolated from other arithmetic operations and paired with its handler.

To illustrate, consider the statement A (I + J) := X * Y;. Notice that the multiplication could generate a Gypsy multiplyerror, and the I + J an adderror. In ADA, both could generate an ADA numeric_error. Similarly the indexing could generate Gypsy indexerror and the assignment a valueerror; both could generate an ADA constraint_error. If the user wrote a handler for adderror after this statement, the indexing operation would have to be performed in the context of the handler and the assignment outside of the handler (so that a range constraint violation on the value of an array item would not be trapped by the handler for the adderror).

This problem was the primary motivating factor in the decision to provide a new version of any Gypsy operation which may raise a condition. Thus the functions for integer arithmetic operations are provided in gypsy_package, and they must be called using function call notation rather than infix

operators. (Integer comparisons are not specially provided, since they may not signal conditions.) These new versions will raise only those exceptions which they could raise in Gypsy. No Ada exceptions will propagate from them into their calling environments. This means that no special treatment is necessary for exception handling at the call site, and that any user-provided handlers may translate directly.

### 2.8.5 Conditional EXIT Specifications

Conditional exit specifications are translated as comments.

## 2.9 Dynamic Objects

Where Gypsy offers several different types of data structures for dynamic objects, the only dynamic form possible in Ada is the linked record structure, where records are tied together with access pointers in a Pascal-like manner. New objects may be allocated, and old ones which have been dropped from linkage with any structure may be de-allocated either explicitly or by the Ada garbage collector. In order to be truly dynamic, all the Gypsy dynamic objects must be implemented as linked record structures.

### 2.9.1 Dynamic Types

Just as any type of data object may be the component type of a dynamic type in Gypsy, any type of data object may be a record field type in the linked record structure. A dynamic type declaration produced by the compiler is a composition of types. The general form for a structured type foo with components of type bar is as follows:

```
type foo is access foo_item;
type foo_item is record f1: bar;
                        f2: foo;
                 end record;
```

The basetype of any dynamic type is the same structure, except that its length is unbounded and its items are of the basetype of the original item type. Since the dynamic types are completely new objects to Ada, the complete set of operations which Gypsy provides on each basetype must be provided by the compiler. The compiler uses a generic package for each kind of dynamic type (set, sequence, mapping) to declare the basetype itself and all of the predefined operations. The predefined statements must also be defined for each basetype, and this is also done in the basetype package.

User types themselves (i.e. with range and element value restrictions) are declared in the packages corresponding to their scope of origin. The type is declared to be an identical subtype of its basetype. Along with each type declaration are constant declarations for the initial value and the typedescriptor for the type. A size restriction on the type is not contained in the type declaration itself; rather it is noted in the typedescriptor, which is provided as a parameter to any predefined operation which must be sensitive to the various restrictions on the type.

#### 2.9.1.1 Sets

Sets are declared as linked records, as described above. The generic package which is instantiated for each set basetype is SET_BASETYPE_ROUTINES. Its parameters include the type kind of the item (an item of the scalar "type type_kind is (g_integer, g_character, g_scalar, g_array, g_record, g_sequence, g_set, g_mapping, g_buffer)") the descriptor of the item basetype, the item basetype name, the name of the equality function for the item, and the name of the valueerror function for the item. It instantiates declarations for the several types, the null value, the typedescriptor for the basetype, all of the predefined operations on the type, and various utility functions which it employs internally. A listing of SET_BASETYPE_ROUTINES may be found in Appendix B.3.

#### 2.9.1.2 Sequences

Sequence types and operations are declared in the same manner as sets. The basetype package is SEQUENCE_BASETYPE_ROUTINES. A listing of this package is in Appendix B.2.

Gypsy may produce objects of three sequence types without the types themselves having been declared. This happens implicitly in range expressions, which are defined to be sequences of the type of the items listed as range bounds. Since the compiler cannot produce such a sequence unless its type has been declared, the three types INTEGER_SEQ, CHARACTER_SEQ, and BOOLEAN_SEQ are instantiated in GYPSY_PACKAGE. This is not seen as excessive overhead, since these are the types of sequences most likely to be declared by the user in a program, anyway.

#### 2.9.1.3 Type STRING

Type string in Ada is specified to be an unbounded array of character. In Gypsy it is predefined to be sequence of character. Having CHARACTER_SEQ as a predefined type, as just stated, allows a predefined target for translating type string. All that remains is the treatment of string literals, and this is accomplished with the type coercion function COERCE_STRING, which is defined in GYPSY_PACKAGE. It takes an Ada string literal as its argument and returns a CHARACTER_SEQ. Any Gypsy string literal will be translated as a call to this function on the literal, i.e. "foo" will translate to gypsy_package.coerce_string("foo").

#### 2.9.1.4 Mappings

Mappings are declared in the same manner as sets and sequences, except that each item includes two data fields, one for the domain item and one of the range. The generic package which instantiates mapping types is MAPPING_BASETYPE_ROUTINES, and it is listed in Appendix B.4.

### 2.9.2 Expressions

Most standard operators defined on dynamic types are supplied in the generic packages for dynamic basetypes in functional and procedural form. The others involve functional decomposition at the call sight.

#### 2.9.2.1 Set and Sequence Values

Set and sequence construction expressions may take two kinds of value lists: a list of expressions separated by commas or a range expression. In the case of the range expression, the operation involves an implicit iteration indexing over the bounds of the range expression, with an adjoin performed with each element. The function performing this operation is generated in-line in the basetype package after the generic instantiation of the basetype support package. In the case of a list of expressions, the operation must be functionally decomposed at the call sight into a composition of adjoin operations. This is necessary, since seqconstructor and setconstructor are n-ary operations and Ada functions must have a predetermined number of arguments.

In either case, the basetype functions which must be called are instantiated for the basetype of the result. This means that if a set or sequence constructor appears in the program, the type of the result (or some type which has as its basetype the type of the result) must have been declared to the compiler. Since a user is unlikely to construct a set or sequence if he has declared no analygous types, this will rarely be a problem. The only exception is the string literal, which is an implicit seqconstructor. Strings, however, are of type CHARACTER_SEQ, which is predefined in the translator. Nevertheless, the user needs to beware of this restriction when using constructors.

#### 2.9.2.2 Component Selectors

Sequence and mapping element selectors and subsequence selectors are defined as functions in their respective basetype packages and instantiated for the basetype of each root name expression.

#### 2.9.2.3 Operators

Each standard operator is defined as a function with correct Gypsy semantics in the basetype package for the appropriate structure.

#### 2.9.2.4 Value Alterations

Value alterations on dynamic types are performed in essentially the same manner as for arrays and records. The basetype packages provide functions to perform specific field alterations and omit alterations. Decompositions are done in-line, exactly as described for arrays and records.

### 2.9.3 Statements

Predefined statements operating on the various dynamic types are defined as procedures in the appropriate basetype package.

#### 2.9.3.1 INSERT Statement

The insert statement is provided as a procedure for sets and mappings. Its arguments are the expression to be inserted, the name of the structure being inserted into, and the typedescriptor of the structure.

There are two procedures provided for inserting into sequences, one for inserting before and one for inserting behind. The arguments to these procedures are the expression being inserted, the name of the structure being inserted into, the index of the component before or behind which the insertion is to be performed, and the typedescriptor of the structure.

#### 2.9.3.2 REMOVE Statement

The remove statement is translated to a call to a generically instantiated procedure. For sets and mappings its arguments are the component being removed and the structure from which it is being removed. For sequences the arguments are the sequence name and the index of the component being removed.

#### 2.9.3.3 MOVE Statement

The move statement is translated into either a call to an insert procedure followed by a call to a remove procedure or a call to an assignment procedure followed by a call to a remove procedure. The proper form is determined by the syntax of the move statement according to the semantics described in the Gypsy 2.1 manual.

## 2.10 Concurrency

Concurrency is not implemented in the compiler. Buffers, however, must be implemented as a means for performing input and output.

### 2.10.1 Buffers

Buffers are declared in a manner similar to the other dynamic types. The major differences are the "io_flag" field in the buffer header which is used to indicate the buffer's use as an input/output device and the "last" field, which points to the oldest item in the buffer. The use of the io_flag was described in detail in Section 2.5.7.2 on input and output. Buffer operations and type declarations are instantiated generically from BUFFER_BASETYPE_ROUTINES, which is listed in Appendix B.5.

### 2.10.2 Operations Restrictions

Operation restrictions are used in semantic checking of the program and have no meaning at runtime. For this reason they are ignored by the compiler.

### 2.10.3 Buffer Parameters

The use of buffers as parameters is checked by the parser. Checks of their special semantics need not be performed by the compiler.

### 2.10.4 Statements

Buffer manipulation statements are implemented only to the extent that they are necessary for input and output and sequential execution.

#### 2.10.4.1 RECEIVE Statement

The receive statement is converted by the compiler into a case statement which keys on the value of the io_flag field of the buffer. If the value of the io_flag is non-zero, a read from the appropriate device is performed. If the value is zero, the procedure which performs a receive on an internal buffer is called. For now this procedure is not intended to be useful for concurrent processing. It is defined to perform the extraction of an item from a buffer in the same manner as an item would be extracted from the end of a sequence. A more detailed explanation of the expansion of a receive statement is in the section on input and output.

#### 2.10.4.2 SEND Statement

The send statement is converted by the compiler into a case statement which keys on the value of the io_flag field of the buffer. If the value of the io_flag is non-zero, a print to the appropriate device is performed. If the value is zero, the procedure which performs a send on an internal buffer is called. For now this procedure is not intended to be useful for concurrent processing. It is defined to perform the insertion of an item into a buffer in the same manner as an item would be inserted at the beginning of a sequence. A more detailed explanation of the expansion of a send statement is in the section on input and output.

#### 2.10.4.3 GIVE Statement

A give statement is translated into an expanded send statement followed by a call to the appropriate remove procedure.

#### 2.10.4.4 Communicating Sequential Processes

This is an issue dealing with concurrency not addressed in the compiler.

### 2.10.5 Concurrent Composition

Concurrency is not addressed in the compiler.

### 2.10.6 Specifications

The special specification forms dealing with buffers are not relevant to non-concurrent programs. Since they are not needed for execution, however, they may be translated transparently as comments whenever they are encountered.

## 2.11 Type Abstraction

Type abstraction is not addressed by the compiler. Abstract type declarations, when encountered, will result in an aborted translation.

# Chapter 3
# COMPILER ROLE AND IMPLEMENTATION

## 3.1 The Gypsy Verification Environment

The Gypsy Verification Environment (GVE) is a large program development environment, implemented in LISP, which brings together tools to aid the programmer in specifying, implementing, and verifying Gypsy programs. The following overview of the GVE is largely based on material in the Gypsy 2.0 Verification Environment 6.0 Users Manual [7].

The source text of Gypsy program units enters the GVE by either being read from file, from a text entry mode in the system monitor, or from a buffer in an editor, EMACS [30], linked to the monitor. The text is first converted into a LISP S-expression form called Prefix [14] by a parser derived from Wilhelm Burger's BOBSW parser generator output [5]. Prefix is an internal form of Gypsy programs which is maintained and used throughout the GVE. After the BOBSW parse, the Prefix undergoes semantic checking and clarification by a parser written by Dwight Hare and enhanced by this author. The parser also maintains symbol tables for system-wide use throughout the verification process.

Program units may be revised or new units added at any time using the same parsing mechanism. Existing units may also be altered through use of either the EMACS editor or a Gypsy structure editor [13]. Prefix may be displayed as Gypsy source by a pretty printing package called infprint [1]. Both infprint and the structure editor were implemented by Dwight Hare.

Altering a program unit may require other units to be re-checked for semantic correctness and verification conditions to be re-proven. The global impact of program changes is tracked by the top level component of the GVE. The top level also keeps track of the overall state of each program unit with regard to its semantic completeness and its verification status. The top level incremental development code was originally implemented largely by Mark Moriconi [22].

The verification process begins with the generation of verification conditions for each eligible unit by the vcgen component. A verification condition is a formula representing the relationship between the formal specifications for a unit and the executable code in the unit. If all the verification conditions for a unit can be proven, then the formal specifications and the code are consistent. Vcgen was written by Richard Cohen and Judith Merriam.

To assist the user in proving verification conditions, the GVE houses a powerful interactive natural deduction theorem prover. The prover was written by W. W. Bledsoe [3,4] and integrated into the GVE by Mabry Tyson and Peter Bruell. A symbolic expression evaluator, xeval, assists vcgen and prover in simplifying Prefix expressions. Xeval returns a normalized symbolic form of its arguments, which can be evaluated either in isolation or with respect to given context and/or program state. Xeval commonly reduces simple verification conditions in vcgen to True so that the user needs to confront only the more complex conditions in the prover.

The Gypsy to Bliss compiler consists of three parts. A compilability checker first determines that the program is completely defined and uses machine-dependent features properly. The second phase is an optimizer, which maps the Gypsy Prefix into an optimized, enhanced, and annotated program tree. The optimizer attempts to suppress runtime checks via proof methods and perform in-line expansion

of routines. Finally the program tree is used by a Gypsy to Bliss compiler to produce a file containing a Bliss image of the program. The Bliss file may be compiled by one of the Bliss compilers and run. The optimizer is being implemented by John McHugh and the Bliss compiler by Lawrence Smith [28].

The Gypsy to Ada compiler, which is the subject of this report, is similar in spirit to the Gypsy to Bliss compiler, in that they both utilize compilers for another high level language. Since a variety of Ada compilers are under development, the code produced by the Gypsy to Ada compiler will be widely portable and usable on a variety of machines in the near future. The compiler is integrated into the GVE with an interface at the system monitor level.

## 3.2 Consideration of Intermediate Program Representations

### 3.2.1 Overview of Gypsy Prefix

The Gypsy parser transforms Gypsy source text into an internal representation known as Prefix. Prefix is a tree structure taking the form of a LISP S-expression. The name "Prefix" is appropriate, since at each level the tree consists of an operator in the CAR, or first position, and the operands follow in the CDR.

Since Prefix is a form which may be conveniently and powerfully manipulated and referenced in a LISP environment, it is used to represent all program units throughout program development in the GVE. The GVE database maintains a Prefix representation for each scope, procedure, function, type, constant, and lemma. This may be selectively extracted for processing. Several forms are saved, each reflecting knowledge available at a different stage of development. Pass1 Prefix is the product of the BOBSW parser. Fullprefix or Pass3witherrors results from semantic checking and expansion of Pass1 Prefix. An abstract syntax prefix, which is an abstract representation which can accommodate annotations, is also kept for each unit. A somewhat formal description of Prefix is available in ICSCA internal documentation [5,14].

### 3.2.2 Choosing an Intermediate Representation

At the outset of the project, it was clear that the Gypsy program would need to be represented in some internal form which captured the semantics of both its Gypsy and Ada incarnations. The natural starting point for the transformation was the Gypsy prefix maintained in the GVE, since it embodied the semantics of the Gypsy program. The question was what form would be best for Ada semantics.

An intermediate representation for Ada programs, Diana [12,24], was under development as a joint project of teams from the University of Karlsruhe, Carnegie-Mellon, Intermetrics, and Softech. Diana was primarily designed with Ada compilers in mind, but was suitable for other purposes such as pretty printing. It was a merger of two previously designed forms, TCOL [36] and AIDA [34], and was potentially a military standard.

With the assumption that there would be pretty printers available for Diana representations, a prefix to Diana translation was considered, but ultimately rejected. The primary reason for the rejection was that Diana is not, in fact, a standard Ada representation. Rather, it is an abstract data type, with its published form of attributed trees serving as an abstract model for the definition of Diana. The standardization applied only to the model, meaning that implementations could take whatever form was appropriate for the implementor. Obviously this would make it much harder to find a pretty printer based on a form convenient to our purposes, and writing a Diana S-expression pretty printer was neither within the scope of this project nor deemed worthy of the effort necessary to do it.

The next candidate for an internal representation was Gypsy prefix itself. The semantic similarities between Gypsy and Ada meant that the structure of the Gypsy prefix would be appropriate. The decision was made to use Prefix as a starting point and to modify it to accommodate different forms introduced during the compilation process. Of the several existing Prefix forms, parser prefix was chosen for the initial implementation. Since the two components most likely to impact the compiler, the parser and infprint, both utilized parser prefix, parser prefix offered

the quickest startup time for the implementation. When the new abstract prefix model currently under consideration is installed throughout the system, the compiler will be retrofitted to work from that new dialect.

## 3.3 Adoption and Syntactic Modification of Infprint

Infprint, as previously stated, is the component of the GVE which "un-parses" Gypsy prefix and presents it as Gypsy source text. Infprint processes prefix explicitly by recursive descent, printing the text as it descends. The descent is driven by knowledge embedded in the component about the various operators. Printing operations require only information locally available in the prefix form and knowledge hard-coded in infprint. Symbol table and other global information about the units being printed is not needed. For the most part, infprint processes the prefix without temporary modification, requiring only indentation and character positioning data which it computes dynamically. The notable exception to this mode is the printing of expressions. Before an expression is actually printed, a pass is made over the prefix during which the print length of each prefix item is appended to the front of each item. This prepassed prefix is then used as the data structure for expression printing.

The cosmetic and structural similarities of Gypsy and Ada, combined with the general cleanness of the infprint component, made infprint very attractive as a tool for the pretty printing of Ada prefix. The prefix traversal and pretty printing structure were already in place. The major remaining problems were 1) the determination of what global information will be needed and the points at which it is required, 2) the prepassing of the prefix to embody the semantic modifications being made to the program, and 3) the adaptation of infprint to print the modified prefix using Ada syntax. Though the modifications of infprint turned out to be so sweeping that only its structure survived, having that structure from the beginning was a great boost. It provided an initial thread of functioning code on which to build the bulk of the compiler.

The general problem-solving strategy was to get a bare bones compiler working, and then expand upon that base until a complete compiler was finished. The obvious first step, then, was to make the modifications to infprint which were purely syntactic in nature. With these modifications made, Gypsy programs which contained only those constructs which have more or less the same form in Ada could be compiled.

Ada and Gypsy have many common features, varying only in syntax. For these features the adaptation of infprint to produce Ada was straightforward. The fundamental control structures and control flow mechanisms of Gypsy are all present in Ada, and are identical except for condition handling, which is somewhat different from Ada exceptions. The concepts of function and procedure are similar in the two languages, and the appearance of their headers, local declarations, and bodies differs only cosmetically. The basic simple types of Gypsy: integer, character, and boolean, are present in Ada. (Gypsy rationals are not compiled because they are intended primarily for specification.) Both the declarations and references to arrays and records are directly translatable. The mapping of Gypsy scopes into Ada packages was achieved largely syntactically, provided packages make no circular references amongst themselves.

Similarly, a number of the constructs which would not translate into Ada, but which are not needed in the runtime image, were easily transformed syntactically into comments. These constructs include all non-validated formal specifications, lemmas, functions for specification only, and most of the constructs dealing with data abstraction. The fact they could be presented for the most part in their original syntax enhanced their usefulness as documentation.

Naturally, there were a great number of syntactic modifications to be made. Enumeration of all these changes would be tedious, but an example which illustrates a great many of them appears in Appendix C.1.

## 3.4 Data Structures

The GVE maintains a database of program units which, among other things, stores the various computed prefix forms for the units. These prefix forms include the Pass1 prefix generated by the BOBSW parser, the result of semantic checking (either Fullprefix or Pass3witherrors), and the abstract syntax representation of the unit. Each of these is saved to reduce the amount of recomputing necessary through verification and incremental development.

The various passes which the compiler makes over the prefix transform it to reflect Ada constructs and semantics. Generating new units and tracking their relationships introduces a certain volatility to the units produced. For example, a re-ordering of the type declarations presented to the compiler in a scope, or a re-ordering of scopes presented might produce a basetype and a set of basetype functions with different names. To track the effects of such changes through repeated compilations would require an incremental development capability which is beyond the current scope of this effort.

Therefore the compiler currently maintains its own database of prepassed units. Each unit in the compiler database is stored under the ADA-PREFIX property in the GVE database. These units may be modified during the course of a compilation, but the life of the database is only the life of the compilation. If a new compilation is ordered, the ADA-PREFIX for all existing units is discarded, and a new database is initialized. The units in the Ada database corresponding to user defined units originate from Fullprefix and are the result of the most recent parse of the unit. Units introduced in the compilation, such as basetypes, typedescriptor and initial value constants, and functions not created in generic packages are also stored in the database. ADA-PREFIX is largely identical in form to FULLPREFIX, but many new properties are introduced, and some expression and statement forms not present in Gypsy prefix may occur. Access to the database units is abstracted through access functions. The Ada database is the main data structure used during the compilation.

Another structure which is critical to the compilation is the structured type map. This is constructed during the global program pass and gives for each type the name and the prefix expansion of its basetype. Basetype units created by the compiler also have entries in the structured type map. The structure aids in linking various declared types to common basetypes where such relationships exist.

A smaller, but important, structure created during the prepassing of a scope is an ordering of units based on their mutual dependencies. If units are declared in this order, there will be no forward references in the code produced. The ordering is determined by producing a dependency tree and passing over the tree laterally.

A global name map is constructed for the purpose of correct printing of unit names. It maps the local Gypsy name of units referenced to atoms which, when printed, will produce a global reference in the Ada program.

## 3.5 Prepasses of Gypsy Prefix

Many semantic modifications of the Gypsy program are necessary to effect the translation into Ada. These are performed, where appropriate, on prepasses through the various structures of the Gypsy program, and on the final printing pass over the prefix. The operations performed at the various levels are described in this section.

### 3.5.1 The Global Program Pass

The entry point to the compiler receives a list of scopes which are to be compiled. The first stage of the compilation is a preliminary pass over these scopes, mainly devoted to gathering information relevant to the compilation of types and to ordering the scopes according to their dependencies.

The first operation is to gather the USED lists of all the scopes, thus determining which scopes refer to which others through named units. A sorted ordering of scopes is calculated on the basis of these USED lists. The ordering is such that a scope will only reference another scope if the other precedes the scope in the ordering. A side effect of this ordering is the detection of circular references among scopes, and of scopes which are referenced but not included in the list of scopes to be compiled.

Both of these conditions are not compilable and result in an aborted compilation.

The ordering determines the order in which the Gypsy scopes will be presented as Ada packages. It also determines the order in which the scopes will be prepassed. During the prepass, the type declarations in each scope are sorted by USED lists, just as the scopes were sorted. Then, in order, each type is examined and its basetype determined. Each type receives an entry in the aforementioned structured type map, with the entry consisting of the expanded TYPDEF of the basetype and the name of the basetype. Naturally, if the basetype has not been previously defined, it must be given a name and an entry in the map. The map, which initially included only the predefined types and basetypes of Gypsy, plus the various primitive sequence types, will by the end of this pass include an entry for every user declared type and every basetype in the program. The global nature of this pass will result in unique basetypes, declared so that any two types with the same basetype will map to the same basetype.

Next, the first new units of the Ada database are created, based on information in the structured type map. Each entry in the map is examined. If it is a user-declared type, its fullprefix is modified to reflect basetype information, and symbols are inserted which will serve as names for the initial value constant and valueerror function, and for the typedescriptor constant where needed. If the entry is a basetype, a prefix representation of the unit is CONS-ed up, reflecting all the information necessary to generate the basetype declaration, and also a generated name for the valueerror function on the basetype.

At this point the basetype package itself is printed. The basetypes are ordered and printed as described in Chapter 2. As each one is generated, items are inserted into the basetype's database entry, indicating the names of the functions generated to perform the predefined operations on the type. These items will then be looked up whenever one of the operations is encountered in the compilation.

If the user has specified a main procedure to be compiled, that procedure is examined and the prefix for a new interface procedure is generated and stored for later printing along with the main procedure. The buffers in the parameter list to the main procedure are examined and a list of the basetypes of their item types is calculated. For each of these basetypes, an Ada SEQUENTIAL_IO package is instantiated at the end of the basetype package. Each buffer parameter then has a corresponding in_out file declared so that a send or receive on the buffer may be compiled as a print or read on the file.

### 3.5.2 Prepassing Scopes

The first operation performed in the prepassing of a scope is an updating of the name map in the prefix for the scope. For each type defined in the scope, an entry for its valueerror function and its initial value constant is inserted into the map. If the type is non-dynamic, its typedescriptor constant is also inserted. The name map will later drive the printing of the units in the scope.

Next, the global name map for the scope is generated from the name map of the scope. The global name map will associate unit names in the scope with atoms which may be printed as unit references. Finally the list of packages which this scope/package will reference is computed for the WITH clause of the package.

The printing of the package follows the pattern described in Chapter 2.

### 3.5.3 Prepassing Units

The distinction between those semantic changes which should be made during the prepassing of a unit and those which may be performed in dynamically on the printing pass is dim. The general philosophy taken has been to defer until the printing pass whenever reasonable, in an attempt to prevent an extra operator-driven traversal of the prefix for the unit.

As a result of this decision, the prepass has been reduced to adapting the Gypsy mechanism for exiting from routines. In Gypsy, a routine is exited normally when execution of the last statement in the main statement list is completed. In Ada, normal execution is completed when a return statement is executed. To this end, return statements are inserted at the end of the main statement list. If the

routine is a function, the declaration of the local variable "result" is made explicit. "Result" is given the type of the function, and the return statement uses it as a parameter.

### 3.5.4 Semantic Modifications During the Print Pass

A myriad of minor syntactic modifications to the program occur during the printing pass. They are too numerous to mention, but a number of more significant semantic and syntactic modifications are worth noting.

Calls to standard functions and procedures must be transformed to calls to the appropriate routines generated for use by objects of the operand basetype. In some cases, as in integer comparison, the Ada infix operator will suffice. In many others, particularly those involving operations on the dynamic structured types and those operations which can raise conditions, the correct call to a generated function must be looked up. The Ada database contains the necessary information to do this. Consider the append operation on objects A and B of type foo, where foo is a sequence of some scalar type with basetype foo_barscope. The compiler first determines the basetype of the type of the operands with the help of a staged symbol table for the unit. Then it looks at the database entry for the basetype to find the name of the append function for objects of that basetype, stored internally as APPEND::FOO_BARSCOPE::BASETYPE_PACKAGE. Finally this function call is transformed into its print form, and the function call printed. It will appear as:

```
... basetype_package.foo_barscope.append (a, b) ...
```

Other significant operations performed during the print pass have already been discussed. Among the most significant of these are the transforming of statements including if-expressions, each clauses, structured component assignments with more than one level of indexing, and the functional decomposition of alteration clauses. Function and procedure calls must be transformed to perform valueerror checking on routine parameters, and buffer operations must be expanded to include the possibility of input/output operations. These various forms are recognized during prefix traversal and the modified forms are fed recursively to the compiler for further semantic transformation. When the new forms are encountered, the printer handles them as Ada syntax.

# Chapter 4
# CONCLUSION

The implementation of the compiler is not yet complete. It is fully operational on features described in Chapters 1-7 of the Gypsy manual (with the few exceptions noted), but the later chapters define some translation problems not yet solved. Condition handling and work on dynamic types is mostly complete, but is not quite finished. Data abstraction is not implemented, and while it has only limited impact on programs at runtime, it has been somewhat difficult to implement in other system components. One of the more difficult problems involved in a complete compilation of Gypsy, however, is the mapping of its concurrency mechanisms into those of Ada. Ada's rendezvous concept differs significantly from Gypsy's more flamboyant message buffers, but preliminary analysis indicates there is sufficient capability in Ada to support the Gypsy concept. While some work remains to be done, the compiler implementation is in an advanced state.

A major disappointment which has hampered development of the compiler has been the lack of an available Ada compiler of sufficient quality to test the translated code. Severe space problems and weaknesses in semantic analysis have to date prohibited compilation of much support code and production of any object code from translation.

Although it would be naive to expect a translation from one high level language to another to be totally clean, I was surprised with the difficulty of translating some constructs. In some cases the results are rather messy and may tend to execute slowly. A large amount of generated code, notably functions generated to operate on structured data types, will likely never be exercised. To produce efficient object code, the Ada compiler will need to perform thorough data flow and call structure analysis. Key optimizations will be possible if the Ada compiler realizes that units generated but never called do not need to be compiled. I hope it will be possible to communicate to a good Ada compiler that many of the compile-time and runtime checks which the Gypsy to Ada compiler performs will supersede the need for the Ada compiler to generate code to do the same work. Given a sufficiently intelligent Ada compiler with appropriate pragmas, however, it should be possible ultimately to generate relatively efficient object code with the Gypsy to Ada compiler.

Production of efficient code, however, was not a main goal of the compiler. The single most important thing that Gypsy can provide to Ada users is formal verification. No thorough tests have yet been run to ensure that the semantics of Gypsy programs are being preserved through compilation. That preservation, however, has been the top priority of this project, and results to date appear to be accurate. In the final analysis, I am confident that program semantics and the validity of the Gypsy proofs, though at some expense of efficiency, will be completely preserved through the compilation. There lies the value and the success of this effort.

# Appendix A
# The Predefined Support Package

```
-- This is the Ada package containing all the predefined
-- Gypsy support code.  It includes generic packages
-- for support of arrays, sequences, sets, mappings,
-- and buffers, but these are listed separately
-- in Appendix B.

package GYPSY_PACKAGE is

-- These are the predefined Gypsy conditions

adderror, aliaserror, caseerror, divideerror, indexerror,
  membererror, minuserror, multiplyerror, negativeexponent,
  nonunique, nopred, nosucc, overscale, powererror,
  powerindeterminate, receiveerror, routineerror, senderror,
  spaceerror, suberror, subtracterror, underscale, valueerror,
  zerodivide: exception;

-- The following declarations together define the typedescriptor
-- type, which is itself a variant record keyed on the type_kind

 IDENTIFIER_LENGTH : constant integer := 30;

 subtype IDENTIFIER is string;

 type TYPE_KIND is (g_integer, g_scalar, g_boolean, g_character,
                    g_array, g_record, g_sequence, g_set,
                    g_mapping, g_buffer);

 type pt_fielddescriptor;

 type typedescriptor (kind: type_kind);

 type TYPEDESCRIPTOR (kind: type_kind) is record
    case kind is
       when g_integer   => i_low: integer := integer'first;
                           i_high: integer := integer'last;
       when g_scalar    => s_low: integer := 0;
                           s_high: integer := integer'last;
       when g_boolean   => b_low: boolean := false;
                           b_high: boolean := true;
       when g_character => c_low: character := character'first;
                           c_high: character := character'last;
       when g_array     => index_type: typedescriptor;
                           a_elem_type: typedescriptor;
       when g_record    => r_items: pt_fielddescriptor;
       when g_sequence
             | g_set    => s_size_restriction: integer;
                           s_elem_type: typedescriptor;
       when g_mapping   => m_size_restriction: integer
                                               := integer'last;
                           domain_type: typedescriptor;
                           rng_type: typedescriptor;
       when g_buffer    => b_size_restriction: integer;
```

```
                        b_elem_type: typedescriptor;
   end case;
end record;

type FIELDDESCRIPTOR (kind: type_kind) is record
   name: identifier;
   field_type: typedescriptor(kind);
   next: pt_fielddescriptor;
end record;

type pt_fielddescriptor is access fielddescriptor;

-- Here are the predefined functions for the simple,
unstructured type

generic
  type DISCRETE_TYPE is (<>);

function MAX (D1, D2: in DISCRETE_TYPE) return DISCRETE_TYPE;

generic
  type DISCRETE_TYPE is (<>);

function MIN (D1, D2: in DISCRETE_TYPE) return DISCRETE_TYPE;

function IFF (X, Y : in BOOLEAN) return BOOLEAN;

function IMP (X, Y : in BOOLEAN) return BOOLEAN;

function BOOLEAN_VALUEERROR (LHS : in TYPEDESCRIPTOR(g_boolean);
                             S : in BOOLEAN)
        return BOOLEAN;

procedure BOOLEAN_ASSIGN (x: in out BOOLEAN;
                          y: in BOOLEAN;
                          x_descriptor:
                               typedescriptor(g_boolean));

function BOOLEAN_EQ (EXP1, EXP2: in BOOLEAN) return BOOLEAN;

function BOOLEAN_PRED (B: in BOOLEAN) return BOOLEAN;

function BOOLEAN_SUCC (B: in BOOLEAN) return BOOLEAN;

function BOOLEAN_SCALE (I: in INTEGER) return BOOLEAN;

BOOLEAN_TYPEDESCRIPTOR: constant typedescriptor :=
       (kind => g_boolean, b_low => false, b_high => true);

function CHARACTER_VALUEERROR (LHS :
                                  in TYPEDESCRIPTOR(g_character);
                               S : in CHARACTER)
        return BOOLEAN;

procedure CHARACTER_ASSIGN
            (x: in out CHARACTER; y: in CHARACTER;
             x_descriptor: typedescriptor(g_character));

function character_eq (char1, char2: in character)
                                        return boolean;
```

```
function CHARACTER_PRED (B: in CHARACTER) return CHARACTER;

function CHARACTER_SUCC (B: in CHARACTER) return CHARACTER;

function CHARACTER_SCALE (I: in INTEGER) return CHARACTER;

CHARACTER_TYPEDESCRIPTOR: constant typedescriptor :=
        (kind => g_character,
         c_low => character'first,
         c_high => character'last);

function INTEGER_EQ (int1, int2: in integer) return boolean;

function INTEGER_ADD (x, y: integer) return integer;

function INTEGER_MULTIPLY (x, y: integer) return integer;

function INTEGER_MINUS (x: integer) return integer;

function INTEGER_SUBTRACT (x, y: integer) return integer;

function INTEGER_POWER (x, y: integer) return integer;

function INTEGER_PRED (x: integer) return integer;

function INTEGER_SUCC (x: integer) return integer;

function INTEGER_VALUEERROR (LHS: typedescriptor(g_integer);
                             x: integer) return boolean;

function foo (TK: type_kind) return type_kind;

procedure INTEGER_ASSIGN (x: in out integer;
                          y: in integer;
                          x_descriptor:
                               typedescriptor(g_integer));

integer_typedescriptor: constant typedescriptor :=
        (kind => g_integer, i_low => integer'first,
         i_high => integer'last);

subtype int is integer range -32768 .. 32767;

int_typedescriptor: constant typedescriptor :=
    typedescriptor'(kind => g_integer, i_low => -32768,
                    i_high => 32767);

int_initial: constant int := 0;

-- Declarations for sequence_basetype_routines,
-- set_basetype_routines, mapping_basetype_routines,
-- buffer_basetype_routines, and array_basetype_routines
-- go here, but may be found in Appendix B

 package integer_seq is new sequence_basetype_routines
        (g_integer,integer_typedescriptor, integer, integer_eq,
        integer_valueerror);

 integer_seq_typedescriptor: constant typedescriptor :=
```

```
   (kind => g_sequence,
    s_size_restriction => integer'last,
    s_elem_type => integer_typedescriptor);

integer_seq_initial: constant integer_seq.basetype := null;

package character_seq is new sequence_basetype_routines
       (g_character, character_typedescriptor, character,
        character_eq, character_valueerror);

character_seq_typedescriptor: constant typedescriptor :=
   (kind => g_sequence,
    s_size_restriction => integer'last,
    s_elem_type => character_typedescriptor);

character_seq_initial: constant character_seq.basetype := null;

function coerce_string (S: string) return
                                    character_seq.basetype;

package boolean_seq is new sequence_basetype_routines
       (g_boolean, boolean_typedescriptor, boolean, boolean_eq,
        boolean_valueerror);

boolean_seq_typedescriptor: constant typedescriptor :=
   (kind => g_sequence,
    s_size_restriction => integer'last,
    s_elem_type => boolean_typedescriptor);

boolean_seq_initial: constant boolean_seq.basetype := null;

end;

package body GYPSY_PACKAGE is

function IFF (X, Y : in BOOLEAN) return BOOLEAN is
 begin
    return (not X) or Y;
 end IFF;

function IMP (X, Y : in BOOLEAN) return BOOLEAN is
 begin
    return (X = Y);
 end IMP;

function MAX (D1, D2: in DISCRETE_TYPE) return DISCRETE_TYPE is
  begin
    if DISCRETE_TYPE'POS(D1) >= DISCRETE_TYPE'POS(D2)
      then return D1;
      else return D2;
    end if;
  end MAX;

function MIN (D1, D2: in DISCRETE_TYPE) return DISCRETE_TYPE is
  begin
    if DISCRETE_TYPE'POS(D1) <= DISCRETE_TYPE'POS(D2)
      then return D1;
      else return D2;
    end if;
  end MIN;
```

```
function BOOLEAN_VALUEERROR (LHS : in TYPEDESCRIPTOR(g_boolean);
                             S : in BOOLEAN)
        return BOOLEAN is
 RESULT : BOOLEAN := FALSE;
 begin
   if (S /= LHS.B_LOW) and (S /= LHS.B_HIGH) then RESULT := TRUE;
        end if
   return RESULT;
 end BOOLEAN_VALUEERROR;

procedure BOOLEAN_ASSIGN (x: in out BOOLEAN;
                          y: in BOOLEAN;
                          x_descriptor:
                            typedescriptor(g_boolean)) is
  begin
    if boolean_valueerror (x_descriptor, y)
      then raise valueerror;
      else x := y; end if;
    return;
  end boolean_assign;

function BOOLEAN_EQ (EXP1, EXP2: in BOOLEAN) return BOOLEAN is
  begin
    return (EXP1 = EXP2);
  end boolean_eq;

function BOOLEAN_PRED (B: in BOOLEAN) return BOOLEAN is
  begin
   if B = false then raise nopred;
     else return false; end if;
  end boolean_pred;

function BOOLEAN_SUCC (B: in BOOLEAN) return BOOLEAN is
  begin
    if b = true then raise nosucc;
     else return true; end if;
  end boolean_succ;

function BOOLEAN_SCALE (I: in INTEGER) return BOOLEAN is
 begin
   if I = 0 then return false;
    elsif I = 1 then return true;
    elsif i < 0 then raise underscale;
    else raise overscale; end if;
 end boolean_scale;

 function CHARACTER_VALUEERROR (LHS :
                                  in TYPEDESCRIPTOR(g_character);
                                S : in CHARACTER)
        return BOOLEAN is
  RESULT : BOOLEAN := FALSE;
  begin
    if S < LHS.C_LOW then RESULT := TRUE;
    end if;
    if S > LHS.C_HIGH then RESULT := TRUE;
    end if;
    return RESULT;
  end CHARACTER_VALUEERROR;
```

```
procedure CHARACTER_ASSIGN (x: in out CHARACTER;
                            y: in CHARACTER;
                            x_descriptor:
                             in typedescriptor(g_character)) is
  begin
    if character_valueerror (x_descriptor, y)
      then raise valueerror;
      else x := y;
     end if;
    return;
  end CHARACTER_ASSIGN;

function character_eq (char1, char2: in character)
                                        return boolean is
  begin
   return (char1 = char2);
  end character_eq;

function CHARACTER_PRED (B: in CHARACTER) return CHARACTER is
  begin
    if B = character'first then raise nopred;
      else return character'pred(b); end if;
  end character_pred;

function CHARACTER_SUCC (B: in CHARACTER) return CHARACTER is
  begin
    if b = character'last then raise nosucc;
      else return character'succ(b); end if;
  end character_succ;

function CHARACTER_SCALE (I: in INTEGER) return CHARACTER is
  begin
   if i < 0 then raise underscale;
     else return character'val(i); end if;
   exception
     when constraint_error => raise overscale;
  end character_scale;

function INTEGER_EQ (int1, int2: in integer) return boolean is
  begin
   return (int1 = int2);
  end integer_eq;

function INTEGER_ADD (x, y: integer) return integer is
  begin
    return (x + y);
   exception
    when numeric_error => raise adderror;
  end integer_add;

function INTEGER_MULTIPLY (x, y: integer) return integer is
  begin
    return (x * y);
   exception
    when numeric_error => raise multiplyerror;
  end integer_multiply;

function INTEGER_MINUS (x: integer) return integer is
  begin
    return (- x);
```

```
   exception
    when numeric_error => raise minuserror;
  end integer_minus;

function INTEGER_SUBTRACT (x, y: integer) return integer is
  begin
    return (x - y);
   exception
    when numeric_error => raise subtracterror;
  end integer_subtract;

function INTEGER_DIV (x, y: integer) return integer is
  begin
    if y = 0
       then raise zerodivide;
       else return (x/y);
    end if;
   exception
    when numeric_error => raise divideerror;
  end integer_div;

function INTEGER_MOD (x, y: integer) return integer is
--
-- This function returns the equivalent of "x - (x div) * y" and
-- calls the translator routines rather than using the Ada mod
-- function in order to raise the appropriate exceptions.  Gypsy
-- has no "moderror" exception.
--
  z: integer;
 begin
    z := integer_div (x, y);
    z := integer_multiply (z, y);
    return (integer_subtract (x, z));
 end;

function INTEGER_POWER (x, y: integer) return integer is
  begin
    if (y < 0) then raise negativeexponent; end if;
    if (x = 0) and (y = 0) then raise powerindeterminate; end if;
    return (x ** y);
   exception
    when numeric_error => raise powererror;
  end integer_power;

function INTEGER_PRED (x: integer) return integer is
  begin
    if (x = integer'first)
      then raise nopred;
      else return (x - 1);
     end if;
  end integer_pred;

function INTEGER_SUCC (x: integer) return integer is
  begin
    if (x = integer'last)
      then raise nosucc;
      else return (x + 1);
     end if;
  end integer_succ;
```

```
function INTEGER_VALUEERROR (LHS: typedescriptor(g_integer);
                             x: integer) return boolean is
  begin
    return (x < LHS.i_low) or (LHS.i_high < x);
  end integer_valueerror;

function foo (TK: type_kind) return type_kind is
  begin
    return type_kind'(g_integer);
  end foo;

procedure INTEGER_ASSIGN (x: in out integer;
                          y: in integer;
                          x_descriptor:
                            typedescriptor(g_integer)) is
  begin
    if integer_valueerror (x_descriptor, y)
      then raise valueerror;
      else x := y;
     end if;
    return;
  end integer_assign;

  function COERCE_STRING (S: string)
                          return character_seq.basetype is
      p, q: character_seq.pt_item_type := null;
    begin
      for ch in reverse S'first..S'last
        loop
          p := new character_seq.item_type
                          (item => character(ch), next => q);
          q := p;
        end loop;
       return character_seq.basetype (q);
    end coerce_string;

  -- The package bodies for sequence_basetype_routines,
  -- set_basetype_routines, mapping_basetype_routines,
  -- buffer_basetype_routines, and array_basetype_routines,
  -- normally found here, may be seen in Appendix B

  end GYPSY_PACKAGE; .
```

# Appendix B
# Predefined Support for Structured Types

## B.1 Support for Arrays

```
-- ************************** ARRAYS ****************************
--
-- ARRAY_BASETYPE_ROUTINES is the package of predefined support
-- for array types.  It is a generic package instantiated for
-- each array basetype in the user program.  It includes a
-- declaration of the basetype and basetype descriptor,
-- functions for equality, accessing via index, field alteration,
-- and valueerror checking, and procedures for field assignment
-- and aggregate assignment.

  generic
     index_kind: type_kind;
     index_type_descriptor: typedescriptor (index_kind);
     type index_type is (<>);
     with function index_type_valueerror_occurs
                      (lhs: typedescriptor(index_kind);
                       i: index_type) return boolean;
     elem_kind: type_kind;
     elem_basetype_descriptor: typedescriptor(elem_kind);
     type elem_basetype is private;
     with function elem_basetype_valueerror_occurs
                      (lhs: typedescriptor(elem_kind);
                       v: elem_basetype) return boolean;

   package ARRAY_BASETYPE_ROUTINES is

      BASETYPE_DESCRIPTOR: constant typedescriptor :=
                  (kind => g_array,
                   index_type => index_type_descriptor,
                   a_elem_type => elem_basetype_descriptor);

      type BASETYPE is array (index_type) of elem_basetype;

      function BASETYPE_EQ (a1, a2: in basetype) return boolean;

      function SELECT_COMPONENT
                (A: basetype;
                 A_descriptor: typedescriptor(g_array);
                 i: index_type) return elem_basetype;

      function ALTERATION (A: basetype;
                           A_descriptor: typedescriptor(g_array);
                           i: index_type;
                           x: elem_basetype) return basetype;

      function VALUEERROR_OCCURS (LHS: typedescriptor(g_array);
                                  A: basetype) return boolean;

      procedure ASSIGN (A1: in out basetype;
                        A2: in basetype;
                        A1_descriptor: in
```

```
                              typedescriptor(g_array));

    procedure ELEMENT_ASSIGN (A: in out basetype;
                  A_descriptor: in typedescriptor(g_array);
                  i: in index_type; x: in elem_basetype);

end ARRAY_BASETYPE_ROUTINES;


package body ARRAY_BASETYPE_ROUTINES is

  function BASETYPE_EQ (a1, a2: in basetype) return boolean is
    begin
      return a1 = a2;
    end basetype_eq;

  function SELECT_COMPONENT (A: basetype;
                      A_descriptor: typedescriptor(g_array);
                      i: index_type) return elem_basetype is
      indexerror: exception;
    begin
      if index_type_valueerror_occurs
            (A_descriptor.index_type, i)
        then raise indexerror;
      end if;
      return A(i);
    end select_component;

    function ALTERATION (A: basetype;
                         A_descriptor: typedescriptor(g_array);
                         i: index_type;
                         x: elem_basetype) return basetype is
      indexerror: exception;
      altered_array: basetype;
    begin
      if index_type_valueerror_occurs
            (A_descriptor.index_type, i)
        then raise indexerror;
      end if;
      altered_array := A;
      altered_array(i) := x;
      return altered_array;
    end alteration;

    function VALUEERROR_OCCURS (LHS: typedescriptor(g_array);
                                A: basetype) return boolean is
    begin
      for i in A'first .. A'last
        loop
          if elem_basetype_valueerror_occurs
                          (LHS.a_elem_type, A(i))
            then return true;
          end if;
        end loop;
      return false;
    end valueerror_occurs;

    procedure ASSIGN (A1: in out basetype;
                      A2: in basetype;
                      A1_descriptor:
```

```
                        in typedescriptor(g_array)) is
      valueerror: exception;
   begin
      if valueerror_occurs(A1_descriptor, A2)
         then raise valueerror;
      end if;
      A1 := A2;
   end assign;

   procedure ELEMENT_ASSIGN (A: in out basetype;
                     A_descriptor: in typedescriptor(g_array);
                     i: in index_type;
                     x: in elem_basetype) is
      indexerror, valueerror: exception;
   begin
      if index_type_valueerror_occurs
                     (A_descriptor.index_type, i)
         then raise indexerror;
      end if;
      if elem_basetype_valueerror_occurs
               (A_descriptor.a_elem_type, x)
         then raise valueerror;
      end if;
      A(i) := x;
   end element_assign;

end ARRAY_BASETYPE_ROUTINES;
```

## B.2 Support for Sequences

```
-- ****************************** SEQUENCES **********************
--
-- SEQUENCE_BASETYPE_ROUTINES is the package of predefined
-- support for sequence types.  It is a generic package
-- instantiated for each sequence basetype in the user program.
-- Sequences are implemented as linked records.  The package
-- includes a declaration of the basetype and basetype
-- descriptor, all the Gypsy functions on sequences, a function
-- for checking valueerror, a null constant, and a POINT_TO_LAST
-- function used internally.  Assignment and REMOVE are defined
-- as procedures.

generic
   kind: type_kind;
   item_basetype_descriptor: typedescriptor(kind);
   type item_basetype is private;
   with function item_basetype_equality
        (v1, v2: item_basetype) return boolean;
   with function item_basetype_valueerror_occurs
       (lhs: typedescriptor(kind); v: item_basetype)
                                        return boolean;

package SEQUENCE_BASETYPE_ROUTINES is

 type pt_item_type;

 BASETYPE_DESCRIPTOR: constant typedescriptor :=
                  (kind => g_sequence,
```

```
                s_size_restriction => integer'last,
                s_elem_type => item_basetype_descriptor);

type BASETYPE is new pt_item_type;

type ITEM_TYPE is record item: item_basetype;
                         next: pt_item_type;
    end record;

type PT_ITEM_TYPE is access item_type;

NULL_VALUE: constant basetype := null;

function SIZE (S: basetype) return integer;

function BASETYPE_EQ (S1, S2: basetype) return boolean;

function BASETYPE_NE (S1, S2: basetype) return boolean;

function ADJOIN_FIRST (S: basetype; x: item_basetype)
                                      return basetype;

function ADJOIN_LAST (S: basetype; x: item_basetype)
                                      return basetype;

function SUB (S1, S2: basetype) return boolean;

function POINT_TO_LAST (S: basetype) return pt_item_type;

function SELECT_ELEMENT (S: basetype; i: integer)
                                 return item_basetype;

function IN_SEQUENCE (S: basetype; item: item_basetype)
                                      return boolean;

function SELECT_SUBSEQUENCE (S: basetype; i, j: integer)
                                      return basetype;

function ALTERATION (S: basetype; i: integer;
                     x: item_basetype) return basetype;

function ALTER_SUBSEQUENCE_ASSIGN (S: basetype; i, j: integer;
                                   E: basetype) return basetype;

function ALTER_PUT_BEFORE (S: basetype; i: integer;
                           y: item_basetype) return basetype;

function ALTER_PUT_BEHIND (S: basetype; i: integer;
                           y: item_basetype) return basetype;

function ALTER_SEQOMIT (S: basetype; i: integer)
                                      return basetype;

function APPEND (S1, S2: basetype) return basetype;

function FIRST (S: basetype) return item_basetype;

function NONFIRST (S: basetype) return basetype;

function LAST (S: basetype) return item_basetype;
```

```
 function NONLAST (S: basetype) return basetype;

 function VALUEERROR_OCCURS (LHS: typedescriptor(g_sequence);
                             S: basetype) return boolean;

 procedure ASSIGN (S1: in out basetype; S2: in basetype;
                   S1_descriptor:
                      in typedescriptor(g_sequence));

 procedure REMOVE (S: in out basetype; i,j: in integer);

end SEQUENCE_BASETYPE_ROUTINES;

package body SEQUENCE_BASETYPE_ROUTINES is

 function ADJOIN_FIRST (S: basetype; x: item_basetype)
                                          return basetype is

   SS1: basetype;
   p: pt_item_type;
  begin
    SS1 := select_subsequence(S, 1, size(S));
    p := new item_type(item => x, next => pt_item_type(SS1));
    SS1 := basetype(p);
    return SS1;
   end adjoin_first;

 function ADJOIN_LAST (S: basetype; x: item_basetype)
                                         return basetype is
 --
 -- Creates a new sequence which is identical to S except that
 -- the element x is adjoined to the rear of the sequence.
 --
    SS1: basetype;
    p, q: pt_item_type;
  begin
    SS1 := select_subsequence(S, 1, size(S));
    p := new item_type(item => x, next => null);
    q := pt_item_type(SS1);
    if q = null
      then SS1 := basetype(p);
      else
        q := point_to_last(SS1);
        q.next := p;
     end if;
    return SS1;
  end ADJOIN_LAST;

 function SIZE (S: basetype) return integer is
 --
 -- Returns the length of the sequence pointed to by S
 --
   p: pt_item_type := pt_item_type(S);
   no_of_elems: integer := 0;
  begin
    loop
      if p = null then return no_of_elems; end if;
      p := p.next;
      no_of_elems := no_of_elems + 1;
     end loop;
```

```
 end size;

function BASETYPE_EQ (S1, S2: basetype) return boolean is
--
-- Checks the equality of two sequences of type basetype.
-- Sequences must match in length and identity of items
-- according to item_basetype_equality.
--
  p1: pt_item_type := pt_item_type(S1);
  p2: pt_item_type := pt_item_type(S2);
 begin
   loop
     exit when p1 = null or p2 = null;
     if not item_basetype_equality(p1.item, p2.item)
       then return false;
       else p1 := p1.next;
            p2 := p2.next;
     end if;
   end loop;
   if p1 = null and p2 = null
      then return true;
      else return false;
   end if;
 end basetype_eq;

function BASETYPE_NE (S1, S2: basetype) return boolean is
--
-- Checks inequality of sequences of basetype.
--
 begin
   return not basetype_eq(S1, S2);
 end basetype_ne;

function SUB (S1, S2: basetype) return boolean is
--
-- Checks if S1 is a subsequence of S2.
--
  p1: pt_item_type := pt_item_type(S1);
  p2: pt_item_type := pt_item_type(S2);
 begin
   loop
     if p1 = null then return true;
       elsif p2 = null then return false;
     end if;
     if item_basetype_equality (p1.item, p2.item)
       then p1 := p1.next;
     end if;
     p2 := p2.next;
   end loop;
 end sub;

function POINT_TO_LAST (S: basetype) return pt_item_type is
--
-- An internal support routine used by several of the routines
-- in this package.  Returns a pointer to the last element
-- of sequence S or the null pointer if S is the empty sequence.
--
  p: pt_item_type := pt_item_type(S);
 begin
   if p /= null
```

```
    then loop
           exit when p.next = null;
           p := p.next;
         end loop;
  end if;
  return p;
end point_to_last;

function SELECT_ELEMENT (S: basetype; i: integer)
                                        return item_basetype is
--
-- Returns the ith element from sequence S. Raises indexerror if
-- i is an improper index for the sequence.
--
  indexerror: exception;
  p: pt_item_type := pt_item_type(S);
  j: integer := 1;
 begin
   if (i < 1 or i > size(S)) then raise indexerror; end if;
     loop
       if j = i then return p.item; end if;
       p := p.next;
       j := j + 1;
     end loop;
 end select_element;

function IN_SEQUENCE (S: basetype; item: item_basetype)
                                        return boolean is
--
-- Returns true iff item appears somewhere in the sequence.
--
  p: pt_item_type := pt_item_type(S);
 begin
   loop
     if p = null then return false;
       elsif item_basetype_equality (p.item, item)
         then return true;
       else p := p.next;
     end if;
   end loop;
 end in_sequence;

function SELECT_SUBSEQUENCE (S: basetype; i, j: integer)
                                        return basetype is
--
-- Creates and returns a sequence which is the subsequence of S
-- indexed by i .. j.
--
   indexerror: exception;
   subseq: basetype;
   p, q: pt_item_type := pt_item_type(S);
   k: integer := 1;
  begin
    if (i > j + 1) or (i < 1) or (i > size(S) + 1)
       or (j < 0) or (j > size(S))
     then raise indexerror;
    end if;
    if i = j + 1 then return basetype(null); end if;
    loop
      exit when k = i;
```

```
      p := p.next;
      k := k + 1;
    end loop;
    q := new item_type(p.all);
    subseq := basetype(q);
    loop
      if k = j
       then q.next := null;
            return subseq;
      end if;
      k := k + 1;
      p := p.next;
      q.next := new item_type(p.all);
      q := q.next;
    end loop;
  end select_subsequence;

function ALTERATION (S: basetype; i: integer;
                     x: item_basetype) return basetype is
--
-- Returns a new sequence which is the Ada translation of
-- the Gypsy construct "S with ([i] := x)"
--
   indexerror: exception;
   new_sequence: basetype;
   k: integer := 1;
   p: pt_item_type;
  begin
    if i < 1 or i > size(S) then raise indexerror; end if;
    new_sequence := select_subsequence(S, 1, size(S));
    p := pt_item_type(new_sequence);
    loop
      if k = i
        then p.item := x;
             return new_sequence;
        else p := p.next;
             k := k + 1;
      end if;
    end loop;
  end alteration;

function ALTER_SUBSEQUENCE_ASSIGN (S: basetype; i, j: integer;
                        E: basetype) return basetype  is
--
-- Creates a new sequence which is the Ada equivalent of the
-- Gypsy "S with ([i..j] := E)". Note that E is a sequence
-- expression of basetype and that the resulting sequence
-- need not be of the same length as S.
--
  S1, S2: basetype;
  E1: basetype := E;
  p: pt_item_type;
 begin
   S1 := select_subsequence (S, 1, i-1);
   S2 := select_subsequence (S, j+1, size(S));
   p := point_to_last (E1);
   if p = null
     then E1 := S2;
     else p.next := pt_item_type(S2);
   end if;
```

```
  p := point_to_last(S1);
  if p = null
    then S1 := E1;
    else p.next := pt_item_type(E1);
  end if;
  return S1;
end alter_subsequence_assign;

function ALTER_PUT_BEFORE (S: basetype; i: integer;
                           y: item_basetype) return basetype is
--
-- Creates a new sequence which is the Ada equivalent of the
-- Gypsy "S with (before [i] := y)"
--
   k: integer := 0;
   p: pt_item_type;
   new_sequence: basetype;
   indexerror: exception;
begin
   if (i < 1) or (i > size(S)) then raise indexerror; end if;
   new_sequence := select_subsequence (S, 1, size(S));
   p := pt_item_type(new_sequence);
   loop
     exit when k = i-1;
     k := k+1;
     p := p.next;
   end loop;
   if p = pt_item_type(new_sequence)
     then new_sequence := basetype(new item_type (y, p));
     else p.next := new item_type (y, p);
   end if;
   return new_sequence;
end alter_put_before;

function ALTER_PUT_BEHIND (S: basetype; i: integer;
                           y: item_basetype) return basetype is
--
-- Creates a new sequence which is the Ada equivalent of the
-- Gypsy "S with (behind [i] := y)"
--
  k: integer := 1;
  p: pt_item_type;
  new_sequence: basetype;
  indexerror: exception;
begin
   if (i < 1) or (i > size(S)) then raise indexerror; end if;
   new_sequence := select_subsequence (S, 1, size(S));
   p := pt_item_type(new_sequence);
   loop
     exit when k = i;
     k := k+1;
     p := p.next;
   end loop;
   p.next := new item_type (y, p.next);
   return new_sequence;
end alter_put_behind;

function ALTER_SEQOMIT (S: basetype; i: integer)
                        return basetype is
--
```

```
-- Creates a new sequence which is the Ada equivalent of the
-- Gypsy "S with (seqomit [i])"
--
   k: integer := 0;
   p: pt_item_type;
   new_sequence: basetype;
   indexerror: exception;
 begin
   if (i < 1) or (i > size(S)) then raise indexerror; end if;
   new_sequence := select_subsequence (S, 1, size(S));
   p := pt_item_type(new_sequence);
   loop
     exit when k = i-1;
     k := k+1;
     p := p.next;
   end loop;
   if k=0 then new_sequence := basetype(p.next);
          else p.next := p.next.next;
   end if;
   return new_sequence;
 end alter_seqomit;

function APPEND (S1, S2: basetype) return basetype is
--
-- Creates the new sequence which is the Ada equivalent of the
-- Gypsy "S1 append S2"
--
  SS1, SS2: basetype;
  p: pt_item_type;
 begin
   SS1 := select_subsequence (S1, 1, size(S1));
   SS2 := select_subsequence (S2, 1, size(S2));
   if SS2 = null
     then return SS1;
     elsif SS1 = null
       then return SS2;
   end if;
   p := point_to_last (SS1);
   p.next := pt_item_type(SS2);
   return SS1;
 end append;

function FIRST (S: basetype) return item_basetype is
--
-- Returns the first item in the sequence.
--
begin
  return select_element (S, 1);
end first;

function NONFIRST (S: basetype) return basetype is
--
-- Creates and returns the "cdr" of the sequence, i.e. all but
-- the first item.
--
 begin
   return select_subsequence (S, 2, size(S));
 end nonfirst;

function LAST (S: basetype) return item_basetype is
```

```
--
-- Returns the last item in sequence S.
--
 begin
   return point_to_last(S).item;
 end last;

function NONLAST (S: basetype) return basetype is
--
-- Creates and returns the sequence which contains all but the
-- last element of S.
--
 begin
   return select_subsequence (S, 1, size(S) - 1);
 end nonlast;

function VALUEERROR_OCCURS (LHS: typedescriptor(g_sequence);
                            S: basetype) return boolean is
--
-- Checks whether sequence S is compatible with the sequence
-- indicated by typedescriptor LHS.  This comparison uses the
-- type valueerror checker form the element types.
--
  p: pt_item_type := pt_item_type(S);
 begin
   if LHS.s_size_restriction < size (S)
     then return true;
   end if;
   loop
     if p = null
       then return false;
       elsif item_basetype_valueerror_occurs (LHS.s_elem_type,
                                              p.item)
         then return true;
       else p := p.next;
     end if;
   end loop;
  end valueerror_occurs;

 procedure ASSIGN (S1: in out basetype;
                   S2: in basetype;
                   S1_descriptor:
                        in typedescriptor(g_sequence)) is
 --
 -- Implements the assignment "S1 := S2". To check that this is
 -- a legal assignment the routine Valueerror-occurs checks
 -- whether S2 is compatible with the type of S1 as indicatedn
 -- by S1's typedescriptor.
 --
   valueerror: exception;
  begin
    if valueerror_occurs (S1_descriptor, S2)
      then raise valueerror;
    end if;
    S1 := select_subsequence(S2, 1, size(S2));
  end assign;

 procedure REMOVE (S: in out basetype; i,j: in integer) is
 --
 -- Removes from S the items indexed by [i .. j]. Note that
```

```
-- this is a procedure and the sequence is updated in place
-- rather than copied.
 indexerror: exception;
 p: pt_item_type := pt_item_type(S);
 k: integer := 1;
begin
  if (i > j + 1) or (i < 1) or (i > size(S) + 1)
       or (j < 0) or (j > size(S))
    then raise indexerror;
  end if;
  if i = j + 1 then return; end if;
  if i = 1
    then loop
           S := basetype(S.next);
           if k = j then return; end if;
           k := k + 1;
         end loop;
  end if;
  loop
    exit when k = i - 1;
    p := p.next;
    k := k + 1;
  end loop;
  loop
    exit when k = j;
    p.next := p.next.next;
    k := k + 1;
  end loop;
 end remove;

end SEQUENCE_BASETYPE_ROUTINES;
```

## B.3 Support for Sets

```
-- ************************** SETS **************************
--
-- SET_BASETYPE_ROUTINES is the package of predefined support
-- for set types.  It is a generic package instantiated for each
-- set basetype in the user program.  Sets are implemented
-- as linked records.  The package includes a declaration of the
-- basetype and basetype descriptor, all the Gypsy functions on
-- sets, a function for checking valueerror, and a null
-- constant. Assignment and REMOVE are defined as procedures.

 generic
   kind: type_kind;
   item_basetype_descriptor: typedescriptor(kind);
   type item_basetype is private;
   with function item_basetype_equality (v1, v2: item_basetype)
                                              return boolean;
   with function item_basetype_valueerror_occurs
      (lhs: typedescriptor(kind); v: item_basetype)
                                              return boolean;

   package SET_BASETYPE_ROUTINES is

      type pt_item_type;
```

```
   BASETYPE_DESCRIPTOR: constant typedescriptor :=
               (kind => g_set,
                s_size_restriction => integer'last,
                s_elem_type => item_basetype_descriptor);

   type BASETYPE is new pt_item_type;

   type ITEM_TYPE is record
      item: item_basetype;
      next: pt_item_type;
   end record;

   type PT_ITEM_TYPE is access item_type;

   NULL_VALUE: constant basetype := null;

   function SIZE (S: basetype) return integer;

   function COPY_SET (S: basetype) return basetype;

   function IN_SET (S: basetype; item: item_basetype)
                                          return boolean;

   function ADJOIN_ELEMENT (S: basetype; x: item_basetype)
                                          return basetype;

   function OMIT (S: basetype; x: item_basetype)
                                          return basetype;

   function SUB (S1, S2: basetype) return boolean;

   function BASETYPE_EQ (S1, S2: basetype) return boolean;

   function BASETYPE_NE (S1, S2: basetype) return boolean;

   function UNION (S1, S2: basetype) return basetype;

   function INTERSECTION (S1, S2: basetype) return basetype;

   function DIFFERENCE (S1, S2: basetype) return basetype;

   function VALUEERROR_OCCURS (LHS: typedescriptor(g_set);
                               S: basetype) return boolean;

   procedure ASSIGN (S1: in out basetype;
                     S2: in basetype;
                     S1_descriptor: in typedescriptor(g_set));

end SET_BASETYPE_ROUTINES;


package body SET_BASETYPE_ROUTINES is

 function SIZE (S: basetype) return integer is
 --
 -- Returns the size of the set.  Assumes that no duplicate
 -- elements occur, something assured by the other routines
 -- in this section.
 --
      p: pt_item_type := pt_item_type(S);
```

```
      no_of_elems: integer := 0;
   begin
      loop
         if p = null then return no_of_elems; end if;
         p := p.next;
         no_of_elems := no_of_elems + 1;
      end loop;
   end size;


function COPY_SET (S: basetype) return basetype is
--
-- Creates a copy of Set S.  Used as internal support in many
-- of the other routines in this package.
--
      SS1: basetype := null;
      p: pt_item_type := pt_item_type(S);
      q: pt_item_type;
   begin
      if p /= null then
         q := new item_type(item => p.item,
                            next => null);
         SS1 := basetype(q);
         loop
            p := p.next;
            exit when p = null;
            q.next := new item_type(item => p.item,
                                    next => null);
            q := q.next;
       end loop;
      end if;
      return SS1;
   end copy_set;

   function IN_SET (S: basetype; item: item_basetype)
                                       return boolean is
   --
   -- Returns true iff item is in set S.
   --
       p: pt_item_type := pt_item_type(S);
    begin
       loop
          if p = null then return false;
          elsif item_basetype_equality (p.item, item)
             then return true;
          else p := p.next;
          end if;
       end loop;
    end in_set;


    function ADJOIN_ELEMENT (S: basetype; x: item_basetype)
                                          return basetype is
    --
    -- Copies set S and adjoins element x to the front if
    -- it is not already in the set, otherwise returns the
    -- copy as is.
    --
       SS1: basetype;
       p: pt_item_type;
```

```
begin
   SS1 := copy_set(S);
   if not in_set(S, x) then
      p := new item_type(item => x,
                         next => pt_item_type(SS1));
      SS1 := basetype(p);
   end if;
   return SS1;
end adjoin_element;


function OMIT (S: basetype; x: item_basetype)
                          return basetype is
--
-- Creates and returns a copy of set S with item x omitted.
-- Assumes that the item appears at most once in the set.
--
   SS1: basetype;
   p: pt_item_type;
begin
   SS1 := copy_set(S);
   p := pt_item_type(SS1);
   if p /= null
     then
       if p.item = x
          then SS1 := basetype(p.next);
          else
             loop
                exit when p = null
                  or else p.next.item = x;
                p := p.next;
             end loop;
             if p /= null
                then  p.next := p.next.next;
             end if;
        end if;
   end if;
   return SS1;
end omit;

function SUB (S1, S2: basetype) return boolean is
--
-- Checks whether S1 is a subset of S2.
--
   p: pt_item_type := pt_item_type(S1);
begin
   loop
      if p = null
         then return true;
         elsif not in_set (S2, p.item)
            then return false;
      end if;
      p := p.next;
   end loop;
end sub;

function BASETYPE_EQ (S1, S2: basetype) return boolean is
--
-- Checks equality of two sets by checking whether each is
-- a subset of the other.
```

```
--
begin
   return sub(S1, S2) and sub(S2, S1);
end basetype_eq;


function BASETYPE_NE (S1, S2: basetype) return boolean is
--
-- Checks inequality of sets S1 and S2.
--
begin
   return not basetype_eq(S1, S2);
end basetype_ne;

function UNION (S1, S2: basetype) return basetype is
--
-- Takes the union of sets S1 and S2.  Elements which
-- appear in both sets appear only once in the union.
--
   SS1: basetype;
   p: pt_item_type := pt_item_type(S2);
   q: pt_item_type;
begin
   SS1 := copy_set(S1);
   loop
     exit when p = null;
     if not in_set(S2, p.item)
        then
           q := new item_type(item => p.item,
                              next => pt_item_type(SS1));
           SS1 := basetype(q);
      end if;
      p := p.next;
   end loop;
   return SS1;
end union;

function INTERSECTION (S1, S2: basetype) return basetype is
--
-- Forms the set which is the intersection of sets S1
-- and S2.
--
   SI: basetype := null;
   p: pt_item_type := pt_item_type(S1);
begin
   loop
      exit when p = null;
      if in_set(S2, p.item)
         then SI := basetype(new item_type (item => p.item,
                              next => pt_item_type(SI)));

      end if;
      p := p.next;
   end loop;
   return SI;
end intersection;

function DIFFERENCE (S1, S2: basetype) return basetype is
--
-- Forms the set which is the set difference of S1 and S2,
```

```
-- i.e., containing those elements which appear and S1 and
-- not in S2.
--
   SD: basetype := null;
   p: pt_item_type := pt_item_type(S1);
begin
   loop
      exit when p = null;
      if not in_set(S2, p.item)
         then SD := basetype(new item_type (item => p.item,
                                next => pt_item_type(SD)));
      end if;
      p := p.next;
   end loop;
   return SD;
end difference;


function VALUEERROR_OCCURS (LHS: typedescriptor(g_set);
                            S: basetype) return boolean is
--
-- Checks whether set S is compatible with the set type
-- indicated by typedescriptor LHS.  This comparison uses
-- the valueerror checker form the element types.
--
   p: pt_item_type := pt_item_type(S);
begin
   if LHS.s_size_restriction < size (S)
      then return true;
   end if;
   loop
      if p = null
         then return false;
         elsif item_basetype_valueerror_occurs
                        (LHS.s_elem_type, p.item)
            then return true;
            else p := p.next;
      end if;
   end loop;
end valueerror_occurs;


procedure ASSIGN (S1: in out basetype;
                  S2: in basetype;
                  S1_descriptor:
                         in typedescriptor(g_set)) is
--
-- Implements the assignment "S1 := S2". To check that this
-- is a legal assignment the routine Valueerror-occurs
-- checks whether S2 is compatible with the type of S1 as
-- indicated by S1's typedescriptor.
--
   valueerror: exception;
begin
   if valueerror_occurs (S1_descriptor, S2)
      then raise valueerror;
   end if;
   S1 := copy_set(S2);
end assign;
```

```
end SET_BASETYPE_ROUTINES;
```

## B.4 Support for Mappings

```
-- ************************ MAPPINGS ***************************
--
-- MAPPING_BASETYPE_ROUTINES is the package of predefined
-- support for mapping types.  It is a generic package
-- instantiated for each mapping basetype in the user program.
-- Mappings are implemented as linked records.  The package
-- includes a declaration of the basetype and basetype
-- descriptor, all the Gypsy functions on mappings, a function
-- for checking valueerror, a null constant, and a COPY
-- function used internally.  Assignment and REMOVE are
-- defined as procedures.

generic
  domain_kind: type_kind;
  domain_basetype_descriptor: typedescriptor(domain_kind);
  rng_kind: type_kind;
  rng_basetype_descriptor: typedescriptor(rng_kind);
  type domain_basetype is private;
  with function domain_basetype_equality
       (v1, v2: domain_basetype) return boolean;
  with function domain_basetype_valueerror_occurs
              (lhs: typedescriptor(domain_kind);
               v: domain_basetype) return boolean;
  type rng_basetype is private;
  with function rng_basetype_equality (v1, v2: rng_basetype)
                                              return boolean;
  with function rng_basetype_valueerror_occurs
              (lhs: typedescriptor(rng_kind);
               v: rng_basetype) return boolean;

  package MAPPING_BASETYPE_ROUTINES is

     type pt_item_type;

     BASETYPE_DESCRIPTOR: constant typedescriptor :=
                 (kind => g_mapping,
                  m_size_restriction => integer'last,
                  domain_type => domain_basetype_descriptor,
                  rng_type => rng_basetype_descriptor);

     type BASETYPE is new pt_item_type;

     type MAPPING_ITEM_TYPE is record
        domain: domain_basetype;
        rng: rng_basetype;
        next: pt_item_type;
     end record;

     type PT_ITEM_TYPE is access mapping_item_type;

     NULL_VALUE: constant basetype := null;

     function SIZE (M: basetype) return integer;
```

```
   function COPY_MAPPING (M: basetype) return basetype;

   function SELECT_MAPPING_VALUE
     (M: basetype; d: domain_basetype) return rng_basetype;

   function INDEX (M: basetype; d: domain_basetype)
                                        return pt_item_type;

   function SUB (M1, M2: basetype) return boolean;

   function BASETYPE_EQ (M1, M2: basetype) return boolean;

   function BASETYPE_NE (M1, M2: basetype) return boolean;

   function UNION (M1, M2: basetype) return basetype;

   function INTERSECTION (M1, M2: basetype) return basetype;

   function DIFFERENCE (M1, M2: basetype) return basetype;

   function ALTERATION (M: basetype; x: domain_basetype;
                        y: rng_basetype) return basetype;

   function ALTER_MAPOMIT (M: basetype; x: domain_basetype)
                                          return basetype;

   function VALUEERROR_OCCURS (LHS: typedescriptor(g_mapping);
                               M: basetype) return boolean;

   procedure ASSIGN (M1: in out basetype;
                     M2: in basetype;
                     M1_descriptor: in typedescriptor);

end MAPPING_BASETYPE_ROUTINES;


package body MAPPING_BASETYPE_ROUTINES is

   function SIZE (M: basetype) return integer is
   --
   -- Returns the size of the mapping (the number of items
   -- in the domain).
   --
      p: pt_item_type := pt_item_type(M);
      no_of_elems: integer := 0;
   begin
      loop
         if p = null then return no_of_elems; end if;
         p := p.next;
      no_of_elems := no_of_elems + 1;
      end loop;
   end size;


   function COPY_MAPPING (M: basetype) return basetype is
   --
   -- Makes a copy of mapping M.
   --
       MM1: basetype := null;
       p: pt_item_type := pt_item_type(M);
```

```
    q: pt_item_type;
begin
   if p /= null then
      MM1 := basetype (new mapping_item_type
                          (domain => p.domain,
                           rng => p.rng,
                           next => null));
      q := pt_item_type(MM1);
      loop
         p := p.next;
         exit when p = null;
         q.next := new mapping_item_type(domain => p.domain,
                                         rng => p.rng,
                                         next => null);
         q := q.next;
    end loop;
   end if;
   return MM1;
end copy_mapping;

function SELECT_MAPPING_VALUE (M: basetype;
                               d: domain_basetype)
                                 return rng_basetype is
--
-- Given a domain value, returns the appropriate range
-- value according to mapping M.
--
   p: pt_item_type := pt_item_type(M);
   indexerror: exception;
 begin
    loop
       if p = null
          then raise indexerror;
          elsif p.domain = d
             then return p.rng;
       end if;
       p := p.next;
    end loop;
 end select_mapping_value;


 function INDEX (M: basetype; d: domain_basetype)
                                   return pt_item_type is
 --
 -- Given a mapping M and a domain value, return a pointer
 -- to the element (domain value, range value, pointer)
 -- triple which has that domain value if there is one,
 -- else the null pointer.
 --
    p: pt_item_type := pt_item_type(M);
 begin
    loop
       if p = null or else p.domain = d
          then return p;
       end if;
       p := p.next;
    end loop;
 end index;
```

```
function SUB (M1, M2: basetype) return boolean is
--
-- Checks whether M1 is a submapping of M2.
--
   p: pt_item_type := pt_item_type(M2);
begin
   loop
      if p = null
         then return true;
         elsif index(M1, p.domain) = null
               or else index(M1, p.domain).rng /= p.rng
            then return false;
      end if;
      p := p.next;
   end loop;
end sub;

function BASETYPE_EQ (M1, M2: basetype) return boolean is
--
-- Checks equality of mappings M1 and M2 by determining
-- whether each is a submapping of the other.
--
begin
   return sub(M1, M2) and sub(M2, M1);
end basetype_eq;


function BASETYPE_NE (M1, M2: basetype) return boolean is
--
-- Checks inequality of mappings M1 and M2.
--
begin
   return not basetype_eq(M1, M2);
end basetype_ne;



function UNION (M1, M2: basetype) return basetype is
--
-- Takes the union of mappings M1 and M2.  If there is a
-- mapping domain value in common between M1 and M2 for
-- which the associated range values differ, the exception
-- nonunique is raised.
--
   MM1: basetype;
   p: pt_item_type := pt_item_type(M2);
   q: pt_item_type;
   nonunique: exception;
begin
   MM1 := copy_mapping(M1);
   loop
     exit when p = null;
     if index(M2, p.domain) = null
        then
           q := new mapping_item_type
                  (domain => p.domain, rng => p.rng,
                   next => pt_item_type(MM1));
           MM1 := basetype(q);
        elsif index(M2, p.domain).rng /= p.rng
           then raise nonunique;
```

```
      end if;
      p := p.next;
   end loop;
   return MM1;
end union;


function INTERSECTION (M1, M2: basetype) return basetype is
--
-- Takes the intersection of mappings M1 and M2.  If there
-- is a mapping domain value in common between M1 and M2
-- for which the associated range values differ, the
-- exception nonunique is raised.
--
   MI: basetype := null;
   p: pt_item_type := pt_item_type(M1);
   nonunique: exception;
begin
   loop
      exit when p = null;
      if index (M2, p.domain) /= null
         then
            if rng_basetype_equality
                  (select_mapping_value (M1, p.domain),
                   select_mapping_value (M2, p.domain))
               then MI := basetype(new mapping_item_type
                                   (domain => p.domain,
                                    rng => p.rng,
                                    next =>
                                     pt_item_type(MI)));
               else raise nonunique;
            end if;
      end if;
      p := p.next;
   end loop;
   return MI;
end intersection;

function DIFFERENCE (M1, M2: basetype) return basetype is
--
-- Takes the difference of the mappings M1 and M2.  If M2
-- is not a submapping of M2, exception suberror is raised.
--
   MD: basetype := null;
   p: pt_item_type := pt_item_type(M1);
   suberror: exception;
begin
   if not sub (M2, M1) then raise suberror; end if;
   loop
      exit when p = null;
      if index(M2, p.domain) /= null
         then MD := basetype (new mapping_item_type
                              (domain => p.domain,
                               rng => p.rng,
                               next => pt_item_type(MD)));
      end if;
      p := p.next;
   end loop;
   return MD;
end difference;
```

```
function ALTERATION (M: basetype; x: domain_basetype;
                     y: rng_basetype) return basetype is
--
-- Creates the mapping which is the Ada equivalent of the
-- Gypsy "M with ([x] := y)"
--
   M1: basetype;
   p: pt_item_type;
begin
   M1 := copy_mapping(M);
   p := index(M1, x);
   if p = null
      then M1 := basetype (new mapping_item_type
                           (domain => x,
                            rng => y,
                            next => pt_item_type(M1)));
      else p.rng := y;
   end if;
   return M1;
end alteration;

function ALTER_MAPOMIT (M: basetype; x: domain_basetype)
                                        return basetype is
--
-- Creates the mapping which is the Ada equivalent of the
-- Gypsy "M with (mapomit [x])"
--
   M1: basetype;
   p: pt_item_type;
   indexerror: exception;
begin
   if index(M, x) = null then raise indexerror; end if;
   M1 := copy_mapping(M);
   p := pt_item_type(M1);
   if p.domain = x
      then M1 := basetype(p.next);
      else
         loop
            exit when p.next.domain = x;
            p := p.next;
         end loop;
         p.next := p.next.next;
   end if;
   return M1;
end alter_mapomit;

function VALUEERROR_OCCURS (LHS: typedescriptor(g_mapping);
                            M: basetype) return boolean is
--
-- Checks whether mapping M is compatible with the mapping
-- type indicated by typedescriptor LHS.  This comparison
-- uses the valueerror checkers for both domain and range
-- types.
   p: pt_item_type := pt_item_type(M);
begin
   if LHS.m_size_restriction < size (M)
      then return true;
   end if;
```

```
      loop
         if p = null
            then return false;
            elsif (domain_basetype_valueerror_occurs
                     (LHS.domain_type, p.domain)
                  or rng_basetype_valueerror_occurs
                     (LHS.rng_type, p.rng))
               then return true;
               else p := p.next;
         end if;
      end loop;
   end valueerror_occurs;

   procedure ASSIGN (M1: in out basetype;
                     M2: in basetype;
                     M1_descriptor: in typedescriptor) is
   --
   -- Implements the assignment "M1 := M2". To check that this
   -- is a legal assignment the routine Valueerror-occurs
   -- checks whether M2 is compatible with the type of S1 as
   -- indicated by M1's typedescriptor.
   --
      valueerror: exception;
   begin
      if valueerror_occurs (M1_descriptor, M2)
         then raise valueerror;
      end if;
      M1 := copy_mapping(M2);
   end assign;

end MAPPING_BASETYPE_ROUTINES;
```

## B.5 Support for Buffers

```
-- ************************ BUFFERS ****************************
--
-- BUFFER_BASETYPE_ROUTINES is the package of predefined support
-- for buffer types.  It is a generic package instantiated for
-- each buffer basetype in the user program.  Buffers are
-- implemented as linked records.  The package includes a
-- declaration of the basetype and basetype descriptor, all the
-- Gypsy functions on buffers, a function for checking
-- valueerror, a null constant, and an assignment function used
-- for parameter passing.  SEND and RECEIVE are defined as
-- procedures.

   generic
      kind: type_kind;
      item_basetype_descriptor: typedescriptor(kind);
      type item_basetype is private;
      with function item_basetype_equality
        (v1, v2: item_basetype) return boolean;
      with function item_basetype_valueerror_occurs
                (lhs: typedescriptor(kind); v2:item_basetype)
                                              return boolean;
      buffer_size_restriction: integer;

   package BUFFER_BASETYPE_ROUTINES is
```

```
type pt_item_type;

BASETYPE_DESCRIPTOR: constant typedescriptor :=
            (kind => g_buffer,
             b_size_restriction => buffer_size_restriction,
             b_elem_type => item_basetype_descriptor);

type BASETYPE is record
   first: pt_item_type;
   last:  pt_item_type;
   io_flag: integer := 0;
  -- io_flag is 0 if the buffer is an internal buffer.  If
  -- its value is non-zero, it represents the ordinal
  -- position of the buffer in the formal arglist to the
  -- procedure specified as main.
end record;

type ITEM_TYPE is record
   item: item_basetype;
   next: pt_item_type;
end record;

type PT_ITEM_TYPE is access item_type;

NULL_VALUE: constant basetype
          := (first => null,
              last => null,
              io_flag => 0);

function SIZE (B: basetype) return integer;

function FULL (B: basetype) return boolean;

function EMPTY (B: basetype) return boolean;

procedure SEND (itm: in item_basetype;
                B_descriptor: in typedescriptor(g_buffer);
                B: in out basetype);

procedure RECEIVE (received_item: in out item_basetype;
                   item_descriptor: typedescriptor(kind);
                   B: in out basetype);

function VALUEERROR_OCCURS (LHS: typedescriptor(g_buffer);
                            B: basetype) return boolean;

procedure ASSIGN (B1: in out basetype;
                  B2: in basetype;
                  B1_descriptor:
                    in typedescriptor(g_buffer));

end BUFFER_BASETYPE_ROUTINES;

package body BUFFER_BASETYPE_ROUTINES is

   function SIZE (B: basetype) return integer is
   --
   -- Returns the current size of the buffer.
   --
```

```
   p: pt_item_type := pt_item_type(B.first);
   no_of_elems: integer := 0;
begin
   loop
      if p = null then return no_of_elems; end if;
      p := p.next;
      no_of_elems := no_of_elems + 1;
   end loop;
end size;

function FULL (B: basetype) return boolean is
begin
  return
   (size (B) = basetype_descriptor.b_size_restriction);
end full;

function EMPTY (B: basetype) return boolean is
begin
   return (B.first = null);
end empty;

procedure SEND (Itm: in item_basetype;
                B: in out basetype) is
--
-- Put item into buffer B.  Since the check for valueerror
-- is made at the call site, no valueerror check is
-- necessary here.  Also if the buffer if full, exception
-- senderror is raised. No provisions are made for
-- concurrency.

   senderror: exception;
begin
   if full (B)
     then raise senderror;
     else B.first := new item_type (item => Itm,
                                    next => B.first);
   end if;
end;

procedure RECEIVE (received_item: in out item_basetype;
                   item_descriptor: typedescriptor(kind);
                   B: in out basetype) is
--
-- Implements a recieve from buffer B into received_item.
-- Since this is similar to an assignment,
-- item_basetype_valueerror_occurs is required to check
-- compatibility.  Also exception receiveerror is raised if
-- the buffer is empty. No provisions are made for
-- concurrency.
   valueerror, receiveerror: exception;
   p: pt_item_type := B.first;
begin
   if empty (B)
      then raise receiveerror;
       elsif item_basetype_valueerror_occurs
          (item_descriptor, B.last.item)
         then raise valueerror;
      else
         received_item := B.last.item;
         if B.first = B.last
```

```
            then B.first := null; B.last := null;
            else
               loop
                  exit when p.next = B.last;
                  p := p.next;
               end loop;
         end if;
      end if;
   end receive;

   function VALUEERROR_OCCURS (LHS: typedescriptor(g_buffer);
                               B: basetype) return boolean is
      p: pt_item_type := B.first;
   begin
      if LHS.b_size_restriction < size (B)
         then return true;
      end if;
      loop
         if p = null
            then return false;
            elsif item_basetype_valueerror_occurs
                     (LHS.b_elem_type, p.item)
               then return true;
               else p := p.next;
         end if;
      end loop;
   end valueerror_occurs;

   procedure ASSIGN (B1: in out basetype; B2: in basetype;
                     B1_descriptor:
                       in typedescriptor(g_buffer)) is
   --
   -- Implements the assignment "B1 := B2". To check that
   -- this is a legal assignment the routine Valueerror-
   -- occurs checks whether B2 is compatible with the type
   -- of B1 as indicated by B1's typedescriptor. Note that
   -- this does not copy B2.
      valueerror: exception;
   begin
      if valueerror_occurs (B1_descriptor, B2)
         then raise valueerror;
      end if;
      B1 := B2;
   end assign;

end BUFFER_BASETYPE_ROUTINES;
```

# Appendix C
# A TOPS-20 Implementation Prelude

```
{ What follows is an example of an implementation
  prelude which the compiler might use to define
  an environment for compilation and execution of
  the user program.  This is one of many possible
  preludes for running under TOPS-20.}

scope tops_20_predefined =
begin

 { The basic configuration of an actual parameter in the user
   call to main is:

     var [in|out]_[char|int|string]_file[1|2|3|4|5] :
       buffer [pending_size_restriction] of
         [character|integer|string] <input>|<output>

   This configuration is achieved by imposed compositions of
   constant and type definitions from the environment scope and
   variable declarations from the environment unit.  Each buffer
   type, except for the tty types, has its size restriction
   declared as a pending (implementation-supplied) constant, and
   each type is used as the type of exactly one variable in the
   environment.  There are five of each variety of composition,
   so the user may have up to five parameters to main of the same
   internal type.  If an environment variable in_char_file1 is
   used as an actual in the user call to main, the implementation
   looks up the buffer restriction on the corresponding formal
   parameter in main and give that value to the constant
   in_char_file1_size.  An unrestricted formal will result in an
   unrestricted actual.  The names of the actuals must be
   selected from the list of variables declared in the
   environment unit and must be unique. }

  type tty_channel = buffer of character;
  type tty_type = record (console: tty_channel <input>;
                          display: tty_channel <output>);

  const in_char_file1_size: integer = pending;
  const in_char_file2_size: integer = pending;
  const in_char_file3_size: integer = pending;
  const in_char_file4_size: integer = pending;
  const in_char_file5_size: integer = pending;

  const out_char_file1_size: integer = pending;
  const out_char_file2_size: integer = pending;
  const out_char_file3_size: integer = pending;
  const out_char_file4_size: integer = pending;
  const out_char_file5_size: integer = pending;

  const in_int_file1_size: integer = pending;
```

```
const in_int_file2_size: integer = pending;
const in_int_file3_size: integer = pending;
const in_int_file4_size: integer = pending;
const in_int_file5_size: integer = pending;

const out_int_file1_size: integer = pending;
const out_int_file2_size: integer = pending;
const out_int_file3_size: integer = pending;
const out_int_file4_size: integer = pending;
const out_int_file5_size: integer = pending;

const in_string_file1_size: integer = pending;
const in_string_file2_size: integer = pending;
const in_string_file3_size: integer = pending;
const in_string_file4_size: integer = pending;
const in_string_file5_size: integer = pending;

const out_string_file1_size: integer = pending;
const out_string_file2_size: integer = pending;
const out_string_file3_size: integer = pending;
const out_string_file4_size: integer = pending;
const out_string_file5_size: integer = pending;

type in_char_file1_type =
              buffer (in_char_file_size1) of character;
type in_char_file2_type =
              buffer (in_char_file_size2) of character;
type in_char_file3_type =
              buffer (in_char_file_size3) of character;
type in_char_file4_type =
              buffer (in_char_file_size4) of character;
type in_char_file5_type =
              buffer (in_char_file_size5) of character;

type out_char_file1_type =
               buffer (out_char_file1_size) of character;
type out_char_file2_type =
               buffer (out_char_file2_size) of character;
type out_char_file3_type =
               buffer (out_char_file3_size) of character;
type out_char_file4_type =
               buffer (out_char_file4_size) of character;
type out_char_file5_type =
               buffer (out_char_file5_size) of character;

type in_int_file1_type =
              buffer (in_int_file1_size) of integer;
type in_int_file2_type =
              buffer (in_int_file2_size) of integer;
type in_int_file3_type =
              buffer (in_int_file3_size) of integer;
type in_int_file4_type =
              buffer (in_int_file4_size) of integer;
type in_int_file5_type =
              buffer (in_int_file5_size) of integer;

type out_int_file1_type =
              buffer (out_int_file1_size) of integer;
type out_int_file2_type =
              buffer (out_int_file2_size) of integer;
```

```
type out_int_file3_type =
              buffer (out_int_file3_size) of integer;
type out_int_file4_type =
              buffer (out_int_file4_size) of integer;
type out_int_file5_type =
              buffer (out_int_file5_size) of integer;

type in_string_file1_type =
              buffer (in_string_file1_size) of string;
type in_string_file2_type =
              buffer (in_string_file2_size) of string;
type in_string_file3_type =
              buffer (in_string_file3_size) of string;
type in_string_file4_type =
              buffer (in_string_file4_size) of string;
type in_string_file5_type =
              buffer (in_string_file5_size) of string;

type out_string_file1_type =
              buffer (out_string_file1_size) of string;
type out_string_file2_type =
              buffer (out_string_file2_size) of string;
type out_string_file3_type =
              buffer (out_string_file3_size) of string;
type out_string_file4_type =
              buffer (out_string_file4_size) of string;
type out_string_file5_type =
              buffer (out_string_file5_size) of string;

Environment TOPS_20 =
 begin

  var tty: tty_type;
  var ttyin: tty_channel <input>;
  var ttyout: tty_channel <output>;

  var in_char_file1: in_char_file1_type <input>;
  var in_char_file2: in_char_file2_type <input>;
  var in_char_file3: in_char_file3_type <input>;
  var in_char_file4: in_char_file4_type <input>;
  var in_char_file5: in_char_file5_type <input>;

  var out_char_file1: out_char_file1_type <output>;
  var out_char_file2: out_char_file2_type <output>;
  var out_char_file3: out_char_file3_type <output>;
  var out_char_file4: out_char_file4_type <output>;
  var out_char_file5: out_char_file5_type <output>;

  var in_int_file1: in_int_file1_type <input>;
  var in_int_file2: in_int_file2_type <input>;
  var in_int_file3: in_int_file3_type <input>;
  var in_int_file4: in_int_file4_type <input>;
  var in_int_file5: in_int_file5_type <input>;

  var out_int_file1: out_int_file1_type <output>;
  var out_int_file2: out_int_file2_type <output>;
  var out_int_file3: out_int_file3_type <output>;
  var out_int_file4: out_int_file4_type <output>;
  var out_int_file5: out_int_file5_type <output>;
```

```
  var in_string_file1: in_string_file1_type <input>;
  var in_string_file2: in_string_file2_type <input>;
  var in_string_file3: in_string_file3_type <input>;
  var in_string_file4: in_string_file4_type <input>;
  var in_string_file5: in_string_file5_type <input>;

  var out_string_file1: out_string_file1_type <output>;
  var out_string_file2: out_string_file2_type <output>;
  var out_string_file3: out_string_file3_type <output>;
  var out_string_file4: out_string_file4_type <output>;
  var out_string_file5: out_string_file5_type <output>;

 end;
end;
```

# Appendix D
# Translation Examples

## D.1 A Translation Involving Only Syntactic Changes

Here is an example of a compilation performed by an early version of the compiler which made mostly transparently syntactic changes in the program text. The Gypsy the scope being compiled, summation_procedure, is given first, followed by the results of running the compiler over the prefix for the scope.

```
scope SUMMATION_PROCEDURE =
begin

  procedure COMPUTE_SUM (A : SMALL_INTEGER_ARRAY;
                         I, J : ARRAY_INDEX;
                         var S : LARGE_INTEGER) =

  begin
     entry (assume I in [1..MAX_INDEX] & J in [1..MAX_INDEX]);
     exit S = ARRAY_SUMMATION (A, I, J);
     var K : INTEGER := I;
     S := 0;
     loop
        if K > J
             then leave
          else assert S = ARRAY_SUMMATION (A, I, K - 1)
                        & K in [I..J]
                        & I in [1..MAX_INDEX]
                        & J in [1..MAX_INDEX];
               ADD_NEXT (S, A[K]);
               K := K + 1
        end
     end
  end;

  function ARRAY_SUMMATION (A : SMALL_INTEGER_ARRAY;
                            P, Q : ARRAY_INDEX) : INTEGER =

  begin
     exit   (P > Q -> RESULT = 0)
          & (   P le Q
             ->   ARRAY_SUMMATION (A, P, Q)
                = ARRAY_SUMMATION (A, P, Q - 1) + A[Q]);


  end;

  procedure ADD_NEXT (var SUM : LARGE_INTEGER;
                      NEXT : SMALL_INTEGER) =

  begin
     exit SUM = SUM' + NEXT;
     SUM := SUM + NEXT
  end;

  const MAX_INDEX : INTEGER = 10;

  const THOUSAND : INTEGER = 1000;
```

```
  const TEN_THOUSAND : INTEGER = 10000;

  type ARRAY_INDEX = INTEGER[1..MAX_INDEX];

  type SMALL_INTEGER = INTEGER[- 1000..THOUSAND];

  type LARGE_INTEGER = INTEGER[- 10000..TEN_THOUSAND];

  type SMALL_INTEGER_ARRAY =
                array (ARRAY_INDEX) of SMALL_INTEGER;

end;

*****************************************************************
The translator yielded the following code for
summation_procedure.
*****************************************************************

package GYPSY_PACKAGE is
end GYPSY_PACKAGE;

with GYPSY_PACKAGE;
package SUMMATION_PROCEDURE is
use GYPSY_PACKAGE;

   MAX_INDEX: constant INTEGER := 10;

   THOUSAND: constant INTEGER := 1000;

   TEN_THOUSAND: constant INTEGER := 10000;

   subtype ARRAY_INDEX is INTEGER range 1..MAX_INDEX ;

   subtype SMALL_INTEGER is INTEGER range -1000..THOUSAND ;

   subtype LARGE_INTEGER is INTEGER range -10000..TEN_THOUSAND ;

   type SMALL_INTEGER_ARRAY is
         array (ARRAY_INDEX) of SMALL_INTEGER;

   procedure COMPUTE_SUM (A : in SMALL_INTEGER_ARRAY;
                          I, J : in ARRAY_INDEX;
                          S : in out LARGE_INTEGER) ;

   function ARRAY_SUMMATION (A : in SMALL_INTEGER_ARRAY;
                             P, Q : in ARRAY_INDEX)
                                      return INTEGER ;
--   Unit is for specifications only
--   exit (P > Q -> RESULT = 0)
--         & (P le Q
--            -> ARRAY_SUMMATION (A, P, Q)
--               = ARRAY_SUMMATION (A, P, Q - 1) + A(Q));
--

   procedure ADD_NEXT (SUM : in out LARGE_INTEGER;
                       NEXT : in SMALL_INTEGER) ;

end;
```

```
package body SUMMATION_PROCEDURE is

  procedure COMPUTE_SUM (A : in SMALL_INTEGER_ARRAY;
                         I, J : in ARRAY_INDEX;
                         S : in out LARGE_INTEGER) is
--    entry (assume I in [range 1..MAX_INDEX] &
--                   J in [range 1..MAX_INDEX]);
--    exit S = ARRAY_SUMMATION (A, I, J);
  K : INTEGER := I;
  begin
     S := 0;
     loop
        if K > J
             then exit;
          else -- assert S = ARRAY_SUMMATION (A, I, K - 1)
--                  & K in [range I..J]
--                  & I in [range 1..MAX_INDEX]
--                  & J in [range 1..MAX_INDEX]      ;
               ADD_NEXT (S, A(K));
               K := K + 1;
        end IF;
     end LOOP;
  end COMPUTE_SUM;

  procedure ADD_NEXT (SUM : in out LARGE_INTEGER;
                      NEXT : in SMALL_INTEGER) is
--    exit SUM = SUM' + NEXT;
  begin
     SUM := SUM + NEXT;
  end ADD_NEXT;

end SUMMATION_PROCEDURE;
```

## D.2 Type Declarations

A critical part of the translation task is the treatment of types. What follows is an annotated translation of the following Gypsy text.

```
scope test = begin
type lower_case = character ['a..'z];
type color = (red, blue, green);
type cool_color = color [blue..green];
type index = integer [1..10];
type array_type = array (index) of lower_case;
type record_type = record (f1: integer; f2: array_type);
type seq_type = sequence (10) of index;
type set_type = set of cool_color;
type map_type = mapping from index to color;
type buf_type = buffer (1) of character;
end;
```

An inescapable observation is the size of the code generated from this small body of Gypsy text. Most of this bulk of this code embodies standard Gypsy functions which must be generated in-line for scalar and record basetypes, which may not be treated generically. Note that the declarations in basetype_package are associated with the basetypes of the

user's types, while the declarations specific to the type itself appear in a package of the same name as the original user scope.

```
with gypsy_package;
package basetype_package is

-- The basetype of array_type and its support functions are
-- declared generically.  An initial value for the basetype
-- follows.

 package array_type_test is
  new gypsy_package.array_basetype_routines
   (gypsy_package.type_kind'(g_integer),
   (kind => gypsy_package.type_kind'(g_integer),
    i_low => 1, i_high => 10),
   integer range 1..10,
   gypsy_package.integer_valueerror,
   gypsy_package.type_kind'(g_character),
   gypsy_package.character_typedescriptor,
   character,
   gypsy_package.character_valueerror);

 array_type_test_initial: constant
    basetype_package.array_type_test.basetype :=
        (array_type_test_initial'range => character'first);

-- The basetype of type color is declared to be identical to
-- the Gypsy type.  Then come declarations for each predefined
-- function on the scalar basetype, a valueerror checker, a
-- typedescriptor for the basetype, and an initial value
-- constant.

 type color_test is (red, blue, green);

 procedure color_test_assign (lhs : in out color_test;
                              rhs : in color_test;
                              lhs_descriptor :
                               in gypsy_package.typedescriptor) ;

 function color_test_pred (s : in color_test) return color_test ;

 function color_test_succ (s : in color_test) return color_test ;

 function color_test_scale (i : in integer) return color_test ;

 function color_test_eq (s1, s2 : in color_test) return boolean ;

 function color_test_valueerror
              (lhs : in gypsy_package.typedescriptor;
               s : in color_test) return boolean ;

 color_test_initial: constant color_test := color_test'(red);

 color_test_typedescriptor:
        constant gypsy_package.typedescriptor :=
         (kind => gypsy_package.type_kind'(g_scalar),
          s_low => 0, s_high => 2);
```

```
-- The basetype of type record_type is declared as a record with
-- basetype fields.  Then come declarations for each predefined
-- function on the scalar basetype, a valueerror checker, a
-- typedescriptor for the basetype, and an initial value
-- constant.  Note that separate functions are needed for each
-- field for field assignment and alteration.

type record_type_test is
       record f1 : integer;
              f2 : basetype_package.array_type_test.basetype;
       end record;

procedure record_type_test_f1_assign
       (rec : in out record_type_test;
        exp : in integer;
        descriptor : in gypsy_package.typedescriptor) ;

procedure record_type_test_f2_assign
       (rec : in out record_type_test;
        exp : in basetype_package.array_type_test.basetype;
        descriptor : in gypsy_package.typedescriptor) ;

function record_type_test_valueerror
       (descriptor : in gypsy_package.typedescriptor;
        exp : in record_type_test) return boolean ;

procedure record_type_test_assign
       (rec : in out record_type_test;
        exp : in record_type_test;
        descriptor : in gypsy_package.typedescriptor) ;

function record_type_test_f1_alteration
       (rec : in record_type_test; exp : in integer)
                    return record_type_test ;

function record_type_test_f2_alteration
       (rec : in record_type_test;
        exp : in basetype_package.array_type_test.basetype)
    return record_type_test ;

function record_type_test_eq (r1, r2 : in record_type_test)
                                              return boolean ;

record_type_test_initial: constant record_type_test :=
       (f1 => 0, f2 => array_type_test_initial);

record_type_test_typedescriptor:
       constant gypsy_package.typedescriptor :=
        (kind => gypsy_package.type_kind'(g_record),
         r_items =>
           new gypsy_package.field-descriptor
                (kind => gypsy_package.type_kind'(g_integer),
                 name => "F1",
                 field_type =>
                        gypsy_package.integer_typedescriptor,
                 next => new gypsy_package.field-descriptor
                         (kind =>
                            gypsy_package.type_kind'(g_array),
                          name => "F2",
                          field_type =>
```

```
        basetype_package.array_type_test.basetype_descriptor,
                            next => null)));

-- The sequence, set, mapping, and buffer basetypes are all
-- declared as generic instantiations of their respective support
-- packages, which are, incidentally, listed in Appendix B.  The
-- initial value constants are required for each basetype.

 package set_type_test is new gypsy_package.set_basetype_routines
  (gypsy_package.type_kind'(g_scalar), color_test_typedescriptor,
   color_test, color_test_eq, color_test_valueerror);

 set_type_test_initial: constant
        basetype_package.set_type_test.basetype :=
             basetype_package.set_type_test.null_value;

 package map_type_test is
        new gypsy_package.mapping_basetype_routines
  (gypsy_package.type_kind'(g_integer),
   gypsy_package.integer_typedescriptor,
   gypsy_package.type_kind'(g_scalar), color_test_typedescriptor,
   integer, gypsy_package.integer_eq,
   gypsy_package.integer_valueerror, color_test,
   color_test_eq, color_test_valueerror);

 map_type_test_initial: constant
        basetype_package.map_type_test.basetype :=
             basetype_package.map_type_test.null_value;

 package buf_type_test is
        new gypsy_package.buffer_basetype_routines
  (gypsy_package.type_kind'(g_character),
   gypsy_package.character_typedescriptor, character,
   gypsy_package.character_eq,
   gypsy_package.character_valueerror, 1);

 buf_type_test_initial: constant
         basetype_package.buf_type_test.basetype :=
             basetype_package.buf_type_test.null_value;

end basetype_package;

package body basetype_package is

 function record_type_test_eq (r1, r2 : in record_type_test)
                                      return boolean is
  result : boolean := false;
  begin
     result := r1 = r2;
     return result;
  end record_type_test_eq;

 function record_type_test_f2_alteration
        (rec : in record_type_test;
         exp : in basetype_package.array_type_test.basetype)
     return record_type_test is
  result : record_type_test;
  begin
     result := rec;
     result.f2 := exp;
```

```
   return result;
end record_type_test_f2_alteration;

function record_type_test_f1_alteration
      (rec : in record_type_test; exp : in integer)
                  return record_type_test is
result : record_type_test;
begin
   result := rec;
   result.f1 := exp;
   return result;
end record_type_test_f1_alteration;

procedure record_type_test_assign
      (rec : in out record_type_test;
       exp : in record_type_test;
       descriptor : in gypsy_package.typedescriptor) is
begin
   if record_type_test_valueerror (descriptor, exp)
        then raise valueerror;
      else rec := exp;
   end if;
   return;
end record_type_test_assign;

function record_type_test_valueerror
       (descriptor : in gypsy_package.typedescriptor;
        exp : in record_type_test) return boolean is
result : boolean := false;
begin
   if gypsy_package.integer_valueerror
     (gypsy_package.extract_fielddescriptor (descriptor, "F1"),
                                             exp.f1)
      or basetype_package.array_type_test.valueerror_occurs
          (gypsy_package.extract_fielddescriptor
                          (descriptor, "F2"),
           exp.f2)
     then result := true;
    else result := false;
   end if;
   return result;
end record_type_test_valueerror;

procedure record_type_test_f2_assign
      (rec : in out record_type_test;
       exp : in basetype_package.array_type_test.basetype;
       descriptor : in gypsy_package.typedescriptor) is
begin
   if basetype_package.array_type_test.valueerror_occurs
       (gypsy_package.extract_fielddescriptor
              (descriptor, "F2"),
        exp)
       then raise valueerror;
      else rec.f2 := exp;
   end if;
   return;
end record_type_test_f2_assign;

procedure record_type_test_f1_assign
       (rec : in out record_type_test;
```

```
        exp : in integer;
        descriptor : in gypsy_package.typedescriptor) is
begin
   if gypsy_package.integer_valueerror
      (gypsy_package.extract_fielddescriptor
        (descriptor, "F1"),
       exp)
     then raise valueerror;
    else rec.f1 := exp;
   end if;
   return;
end record_type_test_f1_assign;

function color_test_valueerror
              (lhs : in gypsy_package.typedescriptor;
                s : in color_test) return boolean is
 result : boolean := false;
 begin
    result := false;
    if color_test'pos (s) < lhs.s_low
         then result := true;
    end if;
    if color_test'pos (s) > lhs.s_high
         then result := true;
    end if;
    return result;
 end color_test_valueerror;

function color_test_eq (s1, s2 : in color_test)
       return boolean is
 result : boolean := false;
 begin
    result := s1 = s2;
    return result;
 end color_test_eq;

function color_test_scale (i : in integer)
       return color_test is
 result : color_test;
 begin
    if i < 0 then raise underscale;
       elsif i > color_test'pos (color_test'last)
          then raise overscale;
       else result := color_test'val (i);
    end if;
    return result;
 end color_test_scale;

function color_test_succ (s : in color_test)
                  return color_test is
 result : color_test;
 begin
    if s = color_test'last then raise nosucc;
       else result := color_test'succ (s);
    end if;
    return result;
 end color_test_succ;

function color_test_pred (s : in color_test)
                  return color_test is
```

```
 result : color_test;
 begin
    if s = color_test'first
          then raise nopred;
       else result := color_test'pred (s);
    end if;
    return result;
 end color_test_pred;

procedure color_test_assign
       (lhs : in out color_test;
        rhs : in color_test;
        lhs_descriptor : in gypsy_package.typedescriptor) is
 begin
    if color_test_valueerror (lhs_descriptor, rhs)
          then raise valueerror;
    end if;
    lhs := rhs;
    return;
 end color_test_assign;

end basetype_package;

with gypsy_package, basetype_package;
package test is

  subtype color is basetype_package.color_test range
   basetype_package.color_test'(red)..
        basetype_package.color_test'(green);

  color_descriptor: constant gypsy_package.typedescriptor :=
        (kind => gypsy_package.type_kind'(g_scalar),
         s_low => 0, s_high => 2);

  color_initial: constant color :=
         basetype_package.color_test'(red);

  subtype index is integer range 1..10;

  index_descriptor: constant gypsy_package.typedescriptor :=
        (kind => gypsy_package.type_kind'(g_integer), i_low => 1,
         i_high => 10);

  index_initial: constant index := 1;

  subtype lower_case is character range  'a'.. 'z';

  lower_case_descriptor: constant gypsy_package.typedescriptor :=
        (kind => gypsy_package.type_kind'(g_character),
         c_low =>  'a',
         c_high => 'z');

  lower_case_initial: constant lower_case :=  'a';

  subtype cool_color is color range
    basetype_package.color_test'(blue)..
          basetype_package.color_test'(green);

  cool_color_descriptor: constant gypsy_package.typedescriptor :=
         (kind => gypsy_package.type_kind'(g_scalar),
```

```
        s_low => 1,
        s_high => 2);

cool_color_initial: constant cool_color :=
      basetype_package.color_test'(blue);

subtype array_type is
      basetype_package.array_type_test.basetype;

array_type_descriptor: constant gypsy_package.typedescriptor :=
      (kind => gypsy_package.type_kind'(g_array),
       index_type => index_descriptor,
       a_elem_type => lower_case_descriptor);

array_type_initial: constant array_type :=
       (array_type_initial'range => lower_case_initial);

subtype buf_type is basetype_package.buf_type_test.basetype;

buf_type_descriptor: constant gypsy_package.typedescriptor :=
      (kind => gypsy_package.type_kind'(g_buffer),
       s_size_restriction => 1,
       s_elem_type => gypsy_package.character_typedescriptor);

buf_type_initial: constant buf_type :=
           basetype_package.buf_type_test.null_value;

subtype map_type is basetype_package.map_type_test.basetype;

map_type_descriptor: constant gypsy_package.typedescriptor :=
      (kind => gypsy_package.type_kind'(g_mapping),
       m_size_restriction => integer'last,
       domain_type => index_descriptor,
       rng_type => color_descriptor);

map_type_initial: constant map_type :=
           basetype_package.map_type_test.null_value;

subtype set_type is basetype_package.set_type_test.basetype;

set_type_descriptor: constant gypsy_package.typedescriptor :=
      (kind => gypsy_package.type_kind'(g_set),
       s_size_restriction => integer'last,
       s_elem_type => cool_color_descriptor);

set_type_initial: constant set_type :=
           basetype_package.set_type_test.null_value;

subtype seq_type is gypsy_package.integer_seq.basetype;

seq_type_descriptor: constant gypsy_package.typedescriptor :=
      (kind => gypsy_package.type_kind'(g_sequence),
       s_size_restriction => 10,
       s_elem_type => index_descriptor);

seq_type_initial: constant seq_type :=
           gypsy_package.integer_seq.null_value;

subtype record_type is record f1 : integer;
                              f2 : array_type; end record;
```

```
 record_type_descriptor:
  constant gypsy_package.typedescriptor :=
  (kind => gypsy_package.type_kind'(g_record),
   r_items =>
    new gypsy_package.field-descriptor
       (kind => gypsy_package.type_kind'(g_integer),
        name => "F1",
        field_type => gypsy_package.integer_typedescriptor,
        next => new gypsy_package.field-descriptor
                 (kind => gypsy_package.type_kind'(g_scalar),
                  name => "F2",
                  field_type => array_type_descriptor,
                  next => null)));

  record_type_initial: constant record_type :=
                (f1 => 0, f2 => array_type_initial);

end;

package body test is

end test;

.
```

## D.3 A Standard Example

This is an example of the full compiler in action on a small Gypsy program which computes factorial. The Gypsy text is given first, followed by the Ada code produced. The translation was to be performed with respect to the environment TOPS-20, given in Appendix C. The user's call to main was:

"main(in_int_file1, out_int_file1)", which results in the device query and linkage described in g_a_supermain. The Gypsy text for the program is:

```
scope factor=
begin

type buftype = buffer (1) of integer;

procedure main (var in_file: buftype <input>;
                var out_file: buftype <output>) =
begin
 var in_value, out_value: integer;
 receive in_value from in_file;
 out_value := fact (in_value);
 send out_value to out_file;
end;

function fact (n:int) : int =
begin
  entry n > 0;
  exit  fact(n) = factorial(n);
  var i:int:=n;
```

```
  result := 1;
  loop
    if i = 1 then leave end;
    assert factorial(n) = result * factorial(i)
           and i > 1 and n > 0;
    result := result * i;
    i := i - 1;
  end;
end;

function factorial (m:int) :int=
begin
   exit (assume factorial(m)=if m=1 then 1 else m*factorial(m-1)
    fi);
end;

end;  {scope factor}

*****************************************************************
The following is the program produced by the compiler.
*****************************************************************

with gypsy_package;
package basetype_package is

 package buftype_factor is
      new gypsy_package.buffer_basetype_routines
  (gypsy_package.type_kind'(g_integer),
   gypsy_package.integer_typedescriptor,
   integer, gypsy_package.integer_eq,
   gypsy_package.integer_valueerror, 1);

 buftype_factor_initial: constant
        basetype_package.buftype_factor.basetype :=
            basetype_package.buftype_factor.null_value;

 package integer_io is new input_output(integer);

 in_file: integer_io.inout_file;

 out_file: integer_io.inout_file;

end basetype_package;

with gypsy_package, basetype_package;
package factor is

  subtype buftype is basetype_package.buftype_factor.basetype;

  buftype_descriptor: constant gypsy_package.typedescriptor :=
        (kind => gypsy_package.type_kind'(g_buffer),
         s_size_restriction => 1,
         s_elem_type => gypsy_package.integer_typedescriptor);

  buftype_initial: constant buftype :=
        basetype_package.buftype_factor.null_value;

  procedure main (in_file, out_file : in out buftype) ;

  function fact (n : in integer) return gypsy_package.int ;
```

```
 function g_a_fact (n : in gypsy_package.int)
                                 return gypsy_package.int ;

-- Unit is for specifications only
-- function factorial (m : in gypsy_package.int)
--                                 return gypsy_package.int

 procedure g_a_main_entry (in_file, out_file : in out buftype) ;

 procedure g_a_supermain ;

end;

package body factor is

 procedure main (in_file, out_file : in out buftype) is
  in_value, out_value : integer := 0;
  begin
    begin
      case in_file.io_flag is
         when 0
              => basetype_package.buftype_factor.receive
                  (in_value,
                   gypsy_package.integer_typedescriptor,
                   in_file);
         when 1
              => declare
                   g_a_temp1 : integer;
                   begin
                     basetype_package.integer_io.read
                      (basetype_package.in_file, g_a_temp1);
                     in_value := g_a_temp1;
                   end;
         when 2
              => declare
                   g_a_temp2 : integer;
                   begin
                     basetype_package.integer_io.read
                       (basetype_package.out_file, g_a_temp2);
                     in_value := g_a_temp2;
                   end;
         when others => raise caseerror;
       end case;
       out_value := fact (in_value);
       if gypsy_package.integer_valueerror
              (buftype_descriptor.b_elem_type, out_value)
          then raise valueerror;
          else case out_file.io_flag is
                 when 0
                      => basetype_package.buftype_factor.send
                          (out_value, out_file);
                 when 1
                      => basetype_package.integer_io.print
                          (basetype_package.in_file, out_value);
                 when 2
                      => basetype_package.integer_io.print
                          (basetype_package.out_file, out_value);
                 when others => raise caseerror;
               end case;
```

```
    end if;
    return;
   end;
  exception
    when others => raise routineerror;
  end main;

function fact (n : in integer) return gypsy_package.int is
 result : gypsy_package.int := 0;
 begin
   begin
    if gypsy_package.integer_valueerror
              (gypsy_package.int_typedescriptor, n)
          then raise valueerror;
    end if;
    result := g_a_fact (n);
    return result;
   end;
  exception
    when others => raise routineerror;
  end fact;

 -- Unit is for specifications only
-- function factorial (m : in gypsy_package.int)
--                              return gypsy_package.int
--   exit case (is normal :
--              (assume factorial (m) =
--                       if m = 1 then 1
--                         else m * factorial (m - 1)
--                       fi));
--

 procedure g_a_main_entry (in_file, out_file : in out buftype) is
  begin
    begin
     in_file.io_flag := 1;
     out_file.io_flag := 2;
     declare
       g_a_temp3, g_a_temp4 : buftype;
       begin
         basetype_package.buftype_factor.assign
               (g_a_temp3, in_file, buftype_descriptor);
         basetype_package.buftype_factor.assign
               (g_a_temp4, out_file, buftype_descriptor);
         main (g_a_temp3, g_a_temp4);
         basetype_package.buftype_factor.assign
               (in_file, g_a_temp3, buftype_descriptor);
         basetype_package.buftype_factor.assign
               (out_file, g_a_temp4, buftype_descriptor);
       end;
     return;
    end;
   exception
     when others => raise routineerror;
   end g_a_main_entry;

 procedure g_a_supermain is
  filename : gypsy_package.string;
  in_file, out_file : buftype := buftype_initial;
  begin
```

```
    begin
      begin
          tty_io.put (" FILE FOR IN_INT_FILE1 ? ");
          tty_io.beep;
          tty_io.get (filename);
          basetype_package.integer_io.open
           (basetype_package.in_file, filename);
        end;
      begin
          tty_io.put (" FILE FOR OUT_INT_FILE1 ? ");
          tty_io.beep;
          tty_io.get (filename);
          basetype_package.integer_io.create
                (basetype_package.out_file, filename);
        end;
      g_a_main_entry (in_file, out_file);
      basetype_package.integer_io.close
               (basetype_package.in_file);
      basetype_package.integer_io.close
               (basetype_package.out_file);
      return;
     end;
   exception
      when others => raise routineerror;
   end g_a_supermain;

 function g_a_fact (n : in gypsy_package.int)
                        return gypsy_package.int is
--    entry n > 0;
--    exit case (is normal : fact (n) = factorial (n));
  i : gypsy_package.int := n;
  result : gypsy_package.int := 0;
  begin
    begin
      gypsy_package.integer_assign
       (result, 1, gypsy_package.int_typedescriptor);
      loop
         if i = 1
              then exit;
         end if;
         -- assert factorial (n) =
         --    result * factorial (i) & i > 1 & n > 0;
         gypsy_package.integer_assign
          (result, gypsy_package.integer_times (result, i),
           gypsy_package.int_typedescriptor);
         gypsy_package.integer_assign
                (i, gypsy_package.integer_difference (i, 1),
                 gypsy_package.int_typedescriptor);
      end loop;
      return result;
     end;
   exception
      when others => raise routineerror;
   end g_a_fact;

end factor;
```

35. Wirth, N. "The Programming Language Pascal." Acta Informatica 1, January, 1971, pp. 35-63.

36. Wulf, W.A., Brosgol, B.M., Newcomer, J.M., Lamb, D.A., Levine, D., Van Deusen, M.S., "TCOL-Ada: Revised Report on an Intermediate Representation for the Preliminary Ada Language", Technical Report CMU-CS-80-105, Carnegie- Mellon University, Computer Science Department, February, 1980.

37. Young, W.D. & Good, D.I. "Generics and Verification in Ada", Proceedings of the ACM Symposium on the Ada Language, Boston, Massachusetts, December, 1980.

38. Good, Donald I., Revised Report on the Language Gypsy Version 2.1, Technical Report in preparation, Institute for Computing Science and Computer Applications, University of Texas at Austin.