CMU-CS-80-148

510.7808
[illegible]
80-148
c. 3

The Charrette Ada Compiler

David Alex Lamb, Andy Hisgen, Jonathan Rosenberg, Mark Sherman,
Martha Borkan*

OCTOBER 1980

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

*Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), in part by ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551, and in part by ARPA Order No. 3341-9, under contract MDA903-79-C-0497.

Table of Contents

List of Figures

1. Introduction and Overview

This report describes the implementation of a compiler for preliminary Ada [15]. The compiler runs on a DECsystem10 or DECsystem20 under TOPS-10 or TOPS-20 and produces code for a VAX-11/780 under UNIX.

This paper is intended primarily for others who intend to implement Ada. The reader is assumed to be familiar with Ada, and with conventional compiler implementation techniques.

Two separate but related efforts are described in this paper. The Front End of the compiler was done at Intermetrics, Incorporated; the Back End was done at Carnegie-Mellon University. Strictly speaking, the name "Charrette" refers only to the Back End, but we have combined reports of the two efforts in order to provide a complete description of the compiler. The Front End is written in Simula [2]; the Back End is written in BLISS-10 [30].

Much of the material for this report was taken from four papers submitted to the Ada Symposium in Boston in December, 1980 [12, 25, 26, 27]. These papers have been reworked to fit together in a unified report. We have also included a number of working notes from the Charrette effort that examine aspects of the compiler not covered by the papers.

The remainder of this chapter introduces the goals of the two projects, the structure of the compiler, and the intermediate languages used in the compiler. Chapter 2 describes the run-time representation of types and variables. Chapter 3 describes the Front End. Chapter 4 describes the phases of the compiler that transform the output of the Front End (TCOL_{Ada}) into a lower-level intermediate language (MIL). Chapter 5 discusses the translation of MIL into assembly language. Chapter 6 contains a number of the Charrette working notes.

All of the Ada program examples in this document use the revised Ada syntax [14].

1.1 Goals and Non-Goals

1.1.1 The Front End

As a part of the Ada test and evaluation effort, Intermetrics developed a program to check the semantic correctness of Ada programs. This was done for several reasons: first, to develop an operational definition of the language semantics against which other translators could be compared; second, to determine the impact of the language rules on the design complexity of such a translator; and third, to understand better how to implement translators that could process Ada programs.

Because changes to Ada were anticipated, the design had to be general enough to handle Ada's semantic rules, and modular enough to incorporate changes in the language as they occurred.

1.1.2 The Back End

The Back End project had a multiplicity of goals. Our primary intentions were to produce a compiler for an extensive subset of Ada in a short period of time, to explore the interactions among language features, and discover where the implementation problems might lie. Despite our desire to get a compiler running quickly, we wanted to produce a clean design. Some of us were interested in eventually exploring strategies for implementing some of the novel aspects of Ada, such as separate compilation, tasking, and generics.

The project was also characterised by a number of things we explicitly decided not to do. We did not intend to produce a polished user-friendly compiler. The project was motivated by compiler implementors and language designers rather than by a user community. This freed us from the need to spend time on those areas that were not strongly related to language implementation issues.

We adopted the following guiding precepts:

- Choose implementation techniques that are as general as possible; avoid schemes that require special case analysis.
- Avoid optimisations unless they can be obtained with little effort.
- Write straightforward compiler code; prefer readability of the compiler sources to efficiency of the running compiler.

1.2 Compiler Structure

The Front End consists of five phases. They are typical of compilers built with the use of parser generation systems. A block diagram of the Front End is shown in Figure 1-1.

The Back End consists of seven phases. The phases can be grouped according to the language they take as input and the language they produce as output. A block diagram of the Back End, illustrating its phases and intermediate languages, is shown in Figure 1-2. The intermediate languages are described in Section 1.3.

The first two phases (TYPEREP and XFORM) transform TCOL_{Ada} into MIL. The next three (BLOCK, ACTREP, and CODE) transform MIL into OBJECT. The seventh (PEEP) performs peephole optimisations from OBJECT form into OBJECT form, and the last (OUTPUT) transforms OBJECT into assembly language.

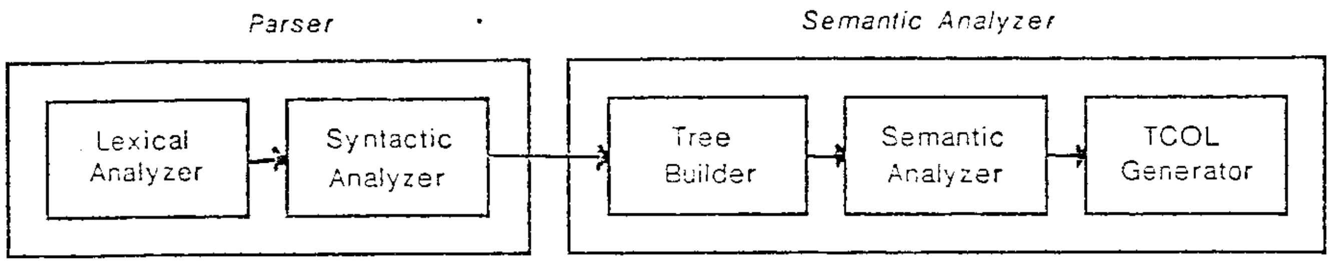


Figure 1-1: Block Diagram of Front End

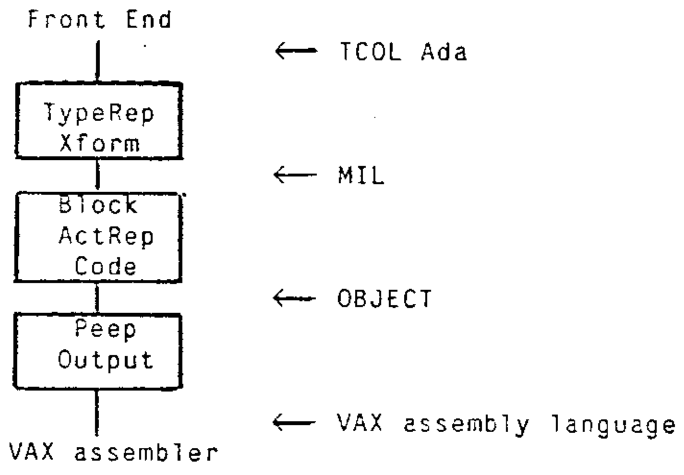


Figure 1-2: Back End Phase Structure

1.2.1 Lexical Analysis

The Lexical Analysis phase used by the Front End is a finite state machine that breaks sequences of characters into tokens. It is automatically generated by a parser generator system.

1.2.2 Syntax Analysis

The Syntax Analysis phase, sometimes termed a parser, is also produced by a parser generator system using an LALR(1) grammar. It produces a list of production rules, identifiers, and literals for use in building a derivation tree.

1.2.3 Tree Builder

The Tree Builder acccepts the list of productions and identifiers and creates a tree that can be manipulated by the semantic analysis phase. This is done by allocating a particular Simula class (variant record) for each rule application and linking these classes (records) together with pointers.

1.2.4 Semantic Analysis

The design of the Front End revolves around the Semantic Analysis phase. This phase performs a tree walk over the program, checking that all declarations, expressions, and statements meet the restrictions in the language reference manual. All binding of names for separately compiled programs, overload resolution, and type checking is done by this phase. Evaluation of static expressions, as required for type checking, is also done at this time.

This phase is described in detail in Chapter 3.

1.2.5 TCOL Generator

The last phase of the Front End, the TCOL Generator, examines the program tree and translates it into an equivalent TCOL format for use by later phases. Conceptually, this phase is the first code generation phase since the original parse tree of the program is discarded and replaced by the simpler TCOL representation.

1.2.6 Typerep

The name Typerep is an abbreviation of "Type Representation". Its purpose is to represent Ada types and subtypes in terms of MIL types. It generates the compiler's internal representation of the descriptors needed at run time.

Typerep performs a recursive walk over the program tree looking for type and subtype information. For each type or subtype, Typerep generates appropriate descriptors for information that must be available at run time.

1.2.7 XFORM

The name XFORM stands for "Tree Transform". This phase has the job of transforming Ada variables, operations, and statements into MIL variables and expressions. By the end of XFORM, all of the original $TCOL_{Ada}$ form of the program has been discarded.

XFORM is described in Chapter 4.

1.2.8 BLOCK and ACTREP

BLOCK's function is to identify the static nesting level of each piece of code in the MIL program. This gives display levels for each variable, identifies the correct exception handler for each context, and locates the targets of **gotos** and **exits**. BLOCK also determines the maximum nesting of blocks within a subprogram, so that a vector of saved stack pointers may be allocated in the activation record for the subprogram (see Section 5.1.3).

The name ACTREP stands for "Actual Representation". In MIL, variables are represented as machine-independent integers and locations. ACTREP decides how much actual storage to allocate for each object and decides on the layout of records and activation records. It also has the responsibility for handling machine-dependent Ada operations and attributes such as `'SIZE`, `'BITS`, and `'POSITION`.

1.2.9 CODE

CODE produces machine code from MIL. It has two portions. One generates code to allocate and initialise declared MIL variables. The other generates machine code from MIL expressions. Both actions are performed in the same tree walk.

CODE treats the VAX as a pure stack machine. This approach eliminates the need to do register allocation, and allows this phase to ignore the context in which an expression occurs when generating code.

CODE is discussed in Chapter 5.

1.2.10 PEEP

PEEP is a simple peephole optimiser; it is one of the few things we did to improve code quality. This phase is driven by a database of optimisation patterns described in a notation much like VAX assembly language. The patterns are converted by a SNOBOL program into BLISS code which manipulates the OBJECT-form data structures.

PEEP is described more fully in a separate paper [19].

1.2.11 Output

Output transforms OBJECT notation into VAX assembly code. It performs a simple graph-to-text translation, and would be easy to modify to produce object files.

1.3 Intermediate Languages

It is possible to view the compiler as a set of transformations between intermediate languages. Each transformation is accomplished by one or more compiler phases. This section gives an overview of the intermediate languages. Discussion of our experiences with these languages is deferred to Section 7.3.1.

1.3.1 LGN

Each of the internal forms of a program is a graph. Nodes within the graph are typed. Each node contains a number of *attribute/value pairs*. The type of a node determines what attributes it may have. For example, in one form there are **tree** nodes representing the parse tree of the program and several kinds of **symbol** nodes representing symbol table information.

In order for a human being to examine these internal structures they must have some textual representation. The notation used is called LGN, for *Linear Graph Notation* [3]. The general shape of the LG form of a node is shown in Figure 1-3. Each node has a label which is denoted by an identifier followed by a colon. Each node also has a type that determines the set of attributes it may have. A portion of a program tree and the corresponding LGN are shown in Figure 1-4.

```
label:  node type
        (attribute-name-1   value(s))
        (attribute-name-2   value(s))
```

Figure 1-3: Characterisation of LGN

1.3.2 TCOL

TCOL$_{Ada}$ is an intermediate language used to represent Ada programs after semantic analysis. It was designed originally for use in the PQCC project [21]. The TCOL form of a program resembles an annotated parse tree together with symbol table information. It differs from a parse tree in that operator identification, type checking, and other aspects of semantic analysis have been performed.

```
                                      :=
                                     /  \
         A := B*C;                  A    *
                                        / \
                                       B   C

L1:      TREE-NODE (OP assign) (SUBNODES L2: L3:)
            TREE-NODE (OP leaf) (DEFN symbol-node-for-A:)
L2:
L3:         TREE-NODE (OP multiply) (SUBNODES L4: L5:)
L4:            TREE-NODE (OP leaf) (DEFN symbol-node-for-B:)
L5:            TREE-NODE (OP leaf) (DEFN symbol-node-for-C:)
```

Figure 1-4: TCOL For a Portion of a Program Tree

TCOL is a rather high-level intermediate form. Its principal advantage is that it retains most of the information provided to the compiler by the source program rather than reducing the program to some simpler form in which much of this information is missing or must be painfully reconstructed.

Figure 1-4 shows the TCOL tree for a simple expression. TCOL_{Ada} is defined in the TCOL_{Ada} report [3]; a partial description is given in a paper by Brosgol [4].

1.3.3 MIL

MIL stands for Machine-independent Intermediate Language; it was designed as part of the Charrette project. Like TCOL it is a tree-based language. It resembles implementation languages like BLISS or C more closely than high-level languages like Ada. Our principal reason for choosing this form as one of the intermediate steps in the compiler was to force a clean separation between the machine-independent portions of the compiler and the machine-dependent ones.

The data structures representable in MIL are integers, pointers, and contiguous blocks of heterogeneous storage. MIL is an expression language; most constructs return a value. MIL retains fairly high-level control constructs such as **if**, **for**, and **case**. This allows the machine-dependent portion of the compiler to chose special instructions intended to be used for such constructs without having to perform extensive analysis of a lower-level representation.

MIL is also discussed in Section 4.1. A complete definition of MIL is given in Appendix VI.

1.3.4 OBJECT

The OBJECT intermediate form is the compiler's internal representation of VAX machine instructions. This form is a doubly-linked list of nodes representing machine instructions; nodes are linked together in the order the instructions will appear in the assembly language program. This intermediate form is based on the data structure used in the FINAL phase of the BLISS-11 compiler [31]. An example of a machine instruction and the corresponding OBJECT form is shown in Figure 1-5.

```
        ADDL2   (SP)+,R0

L1:     OBJECT (MOP ADDL2)(OPERAND L2: L3:)(PREV L0:) (NEXT L4:)
L2:         ADDRESS (MOP AutoIncrement)(OPERAND SP)
L3:         ADDRESS (MOP Register) (OPERAND R0)
```

Figure 1-5: OBJECT Form of a VAX Machine Instruction

1.3.5 VAX Assembly Language

We compile to assembly language primarily to avoid worrying about the details of object file formats. This also allows us to transfer text files to the target machines, avoiding problems of compatibility of binary file formats.

2. A Run-time Representation for Ada Variables and Types

This chapter presents a run-time representation for Ada variables and types. The type and subtype facilities of the Ada programming language permit some subtype information to be determined dynamically. This subtype information requires a run-time representation, and its dynamic nature influences the representation of variables. In this chapter, we first review Ada's types and subtypes to identify some of those aspects that affect run-time representation. We then present the particular representation scheme used in the Charrette Ada implementation. The scheme is straightforward and consistent in that a variable is represented the same way independently of whether it is on the stack, on the heap, or a component of another variable. The design treats Ada's discriminants and discriminant constraints as a form of parameterised types, where the parameterisation permits different instances of a type to have different variants and different sizes for array fields. Composition of such parameterised types is supported. We explain how several Ada operations are handled by our particular representation. We briefly discuss some alternative approaches to Ada representation, comparing them to our design.

2.1 Review of Ada

This section reviews some aspects of Ada that impact run-time representations [15, 17].

2.1.1 Constraints and Subtypes

The Ada language has the notions of *type* and *subtype*. A subtype is a type with *constraints*. Two variables may have the same type, but different constraints, and thus, different subtypes. Constraints may be run-time expressions; thus, in general, if an operation requires subtypes to match, the checking must be done at run time.

There are several kinds of constraints. Scalar types may have *range constraints*. Arrays have *index constraints*, which determine the array bounds. These bounds must be checked upon subscripting and slicing and upon assigning and comparing entire arrays. Slices likewise have index constraints. Records may have *discriminant* fields which are used as the array bounds on another field in the same record or as the tag on a variant part. A record's discriminants may be restricted to specific values by a *discriminant constraint*. Access types may have either index constraints or discriminant constraints, depending on whether the accessed object is an array or a record, respectively.

In addition to requiring run-time checking, Ada permits a programmer to make *attribute inquiries*. For example, the user may obtain the lower bound of an array via the `'FIRST` attribute.

2.1.2 Dynamic Arrays Within Records

In much of Ada, the subtype constraints for a particular variable are fixed upon allocation of the variable. This holds for allocation on the stack upon block entry and for allocation on the heap by **new**. One exception occurs with an unconstrained record variable having a discriminant that is used as an array bound. The variable Z in the example below is such a record.

```
subtype Natural is Integer range 1..Integer'LAST; -- repeated from
type String is array (Natural range <>) of Character; -- package STANDARD

type MyVStr(Len: Integer range 1..133 := 120) is
  record
    SVal: String(1..Len);
  end record;

X: MyVStr(Len => 39);
Y: MyVStr(Len => 15);
Z: MyVStr;          -- Unconstrained.  The discriminant
                    -- Len may take on any value in the range
                    -- 1..133, and the array field SVal may
                    -- become as big as 133 elements.  Initially,
                    -- Len is 120 and SVal has bounds 1..120.
begin
Z := Y;
Z := X;
X := Z;          -- Requires check that Z.Len = 39
...
```

Since it has no discriminant constraint, the record variable Z may have its discriminant Len changed by an entire record assignment; this also changes the bounds (i.e., the index constraints) on the array field SVal. Thus, we require some *changeable* association of the array variable with its subtype constraints.

In the above discussion, we made such dynamic arrays within records appear to be an exceptional case in that their constraints changed after creation of the variable. However, one may view an entire record assignment that changes discriminants as re-creating the component variables. This re-creation view is especially appropriate for an entire record assignment that changes variants, for such a change conceptually destroys the component fields belonging to the old variant while creating the components that belong to the new variant.

Implementations may strive for space efficiency in representing the constrained instances of a type (e.g., the variables X and Y above). We can mix constrained and unconstrained instances of a record in assignments, with checking in the appropriate cases. We can view such mixing as involving mild changes of representation.

2.1.3 Heap Objects and Constraints

For access types, the user may defer supplying index and discriminant constraints for the accessed type until the object is actually allocated via **new**. For example,

```
type PtrString is access String;
P: PtrString;
begin
P := new String(1..4000);
P := new String(M..N);
...
```

In general, the subtype constraint information for each heap object must be associated with that particular object, to make it feasible to access this information via a pointer to the object.

2.1.4 Subtype Constraints and Formal Parameters of Subprograms

For unconstrained array formal parameters, the formal inherits the bounds of the actual array parameter. Thus, the bounds of the actual must be available throughout the call on the subprogram.

For unconstrained record formal parameters, the formal initially has the discriminant values of the actual. If the actual is constrained, then so is the formal. Furthermore, the subprogram body may inquire whether the actual parameter is constrained by applying the 'CONSTRAINED attribute to the formal parameter.

2.1.5 Composition of Discriminants and Discriminant Constraints

In preliminary Ada, a discriminant of a record could only be used as an array bound or as a tag on a variant part. In revised Ada, a discriminant may also be used in a discriminant constraint on another field of the record. This feature permits composition of records with composition of their discriminant constraints, since a discriminant constraint on an outer record may be used in a constraint on an inner record. This allows two instances of the same outer record type to have different constraints on an inner record field, for example, the variables ATS and BTS below have different constraints on their S1 field.

```
type TwoStr(N1,N2: Integer range 1..512) is
  record
    S1: MyVStr(Len => N1);
    S2: MyVStr(Len => N2);
  end record;
ATS: TwoStr(N1 => 39, N2 => 23);
BTS: TwoStr(N1 => 15, N2 => 23);
```

Ada's discriminant and discriminant constraint mechanisms can be regarded as a form of parameterised types, where the parameterisation permits different instances of a type to have different variants and different sizes for array fields [10].

2.2 A Particular Representation

This section describes our run-time representation scheme. We first give overviews of the representations of types, subtypes, and variables, and then fill in the details in later sections. Section 4.2 covers the translation of the Ada source program into these representations.

2.2.1 Overview of Representation of Types and Subtypes

As explained in Section 2.1.5, Ada's discriminants and discriminant constraints may be viewed as type parameterisation. One may also view array types and index constraints as being a parameterisation of arrays; the index constraints are the parameters. We employ this view in our implementation. Each non-scalar type definition in the Ada source program is described at run-time by a *type template*. A type template contains a tag to indicate whether it is for an array, record, or access type. A type template for an array or record may take formal parameters. The formal parameters of a record type template are simply its discriminants. The formal parameters of an array type template are the bounds for the array plus any parameters to the element type.

When a type is used in an Ada source program, constraints may be supplied. In our run-time representation, these constraints become actual parameters to a type template.

```
CRTag: one of CRConstrained, CRUnconstrained,
       or CRParentsParam
----------------------------------------------
CRValue: depends on CRTag:
when CRConstrained, scalar value of constraint
when CRUnconstrained, pointer (absolute) to
     scalar type descriptor for this type (if
     known), or nil (if unknown),
when CRParentsParam, index of parent's
     parameter to use, e.g., use the third
     parameter of my parent.
```

Figure 2-1: Constraint-rep-block

An actual type parameter is represented by an entity called a *constraint-rep-block* (see Figure 2-1). It handles three kinds of actual type parameters: (1) constrained, (2) unconstrained, and (3) constrained by a type parameter of the enclosing variable (the *parent*). A constraint-rep-block has two fields: a tag indicating which of the three cases exists and a value whose interpretation depends on this tag.

All the type actuals in an index or discriminant constraint are packaged up in an entity called a *call block*. A call block may be thought of as an actual argument list. It also contains a pointer to the type template to which it is being applied, and a count of the number of type actuals. Thus, a subtype formed by an index or discriminant constraint is represented by a call block.

A record has component fields, each having its own subtype. For each field, the type template for a record type contains a pointer to the template or call block for that field. An array type template contains a pointer to the template or call block for the component type.

Some of the goals of this parameterised treatment of Ada types are:

- Handling parameterisation and composition. For an instance of a composite type, actual parameters must be passed down from the top-level type into the component types.
- Handling unconstrained instances of types. Just as other actual parameters must be passed down, the special parameter value "unconstrained" must also be passed down so that the correct amount of storage will be allocated for the components.
- Handling discriminants being used in discriminant constraints on other fields.
- Handling a constrained instantiation of a parameterised type T nested inside another type T2 when this constraint is not a parameter of T2. For example,

  ```
  type T2 is record A: T(3,M); end record;
  ```

  (Here, the variable M is from a surrounding scope.)

2.2.2 Overview of Representation of Variables

Constraint information is stored in every instance of an array or record variable. Each array variable has its own copy of the index constraints (the bounds) and each record variable has its own copy of the discriminants, plus a single bit that tells if it is constrained.

The representation of both array and record variables consists of two parts, the *fixed part* and the *dynamic part*. The fixed part has a size which is known at compile time, and is thus allocated at a manifest (i.e., compile-time-known) offset in the enclosing record or stack frame. The dynamic part has a size which must be determined at run time. The fixed part of a variable contains a pointer (as an offset) to the corresponding dynamic part.

Our representation uses offsets for internal pointers within variables rather than absolute addresses.

Records are stored contiguously: the fixed parts for the fields is immediately followed by the dynamic parts. These implementation decisions are intended to facilitate using block copying for assignment. Our original motivation for block copies was that the preliminary Ada Language Reference Manual [15] and Rationale [16] imply that block copies are feasible. We wanted our design process to check these implicit claims. As our design and implementation have progressed, we have never encountered a reason for abandoning these representations.

2.2.3 Scalar Subtypes

The descriptor for a scalar subtype is shown in Figure 2-2. This descriptor also handles scalar types, as a scalar type `T` has a range of `T'FIRST` through `T'LAST`. The tag may be either of two values, *StatScalar* or *DynScalar*. If the tag is *DynScalar*, then the range is run-time determined, and variables of this subtype are given a liberal allocation of one VAX longword (4 bytes). If the tag is *StatScalar*, then the range was compile-time determined, and variables are given only as many bytes as this range requires. The motivation for this special-casing is to allocate strings as one character per byte; this is desired both for efficient storage utilisation and for compatibility with other software on the VAX.

```
Tag: one of StatScalar or DynScalar
-------------------------------------------
Low: 'FIRST for subtype
-------------------------------------------
Hi: 'LAST for subtype
```

Figure 2-2: Scalar Subtype Descriptor

2.2.4 Arrays

The type descriptor for an array type is shown in Figure 2-3.

An array variable has two parts, fixed and dynamic. The fixed part is a fairly conventional array descriptor, as shown in Figure 2-4. The dynamic part is the storage for the array elements themselves. The *VirtualZeroOrigin* field of the fixed part is an offset to the array elements which has had the lower bounds already subtracted off. For an n-dimensional array, we store n-1 multipliers. The extra multiplier field is used to store the size of an individual element. (These techniques are discussed in Gries [9].)

We represent a slice as an array variable fixed part that has no dynamic part of its own. Instead, its

```
Tag: always ArrayTemplateTag
-----------------------------------------------
NFParams: number of formal type parameters to
     this type
-----------------------------------------------
NDims: number of dimensions
-----------------------------------------------
EltDesc:  pointer (absolute) to type or
     subtype descriptor for the element type
-----------------------------------------------
IndexTyp: pointer (absolute) to the scalar
     subtype descriptor for the index type
     for this dimension
(Repeated for each dimension)
```

Figure 2-3: Array Type Descriptor

fixed part points into the middle of some other array variable's elements. This means that slices look like ordinary array variables. We are able to take a slice of an array and then pass that slice as an actual parameter without copying any elements. If our array variable representation were split such that array bounds were assumed to be adjacent to the elements, then slicing would require some copying of elements for the slice to become an ordinary array variable.

```
Tag: always ArrayVariableTag
-----------------------------------------------
OffsetBase: pointer (offset) to base of
     dynamic part
-----------------------------------------------
NDims: number of dimensions
-----------------------------------------------
TotalESize: total size of elements
-----------------------------------------------
Lower Bound          (These 3 fields are
----------------      repeated NDims times)
Upper Bound
----------------
Multiplier
-----------------------------------------------
VirtualZeroOrigin: offset to virtual origin
     of element (0,0,...,0)
```

Figure 2-4: Array Variable, Fixed Part

2.2.5 Records

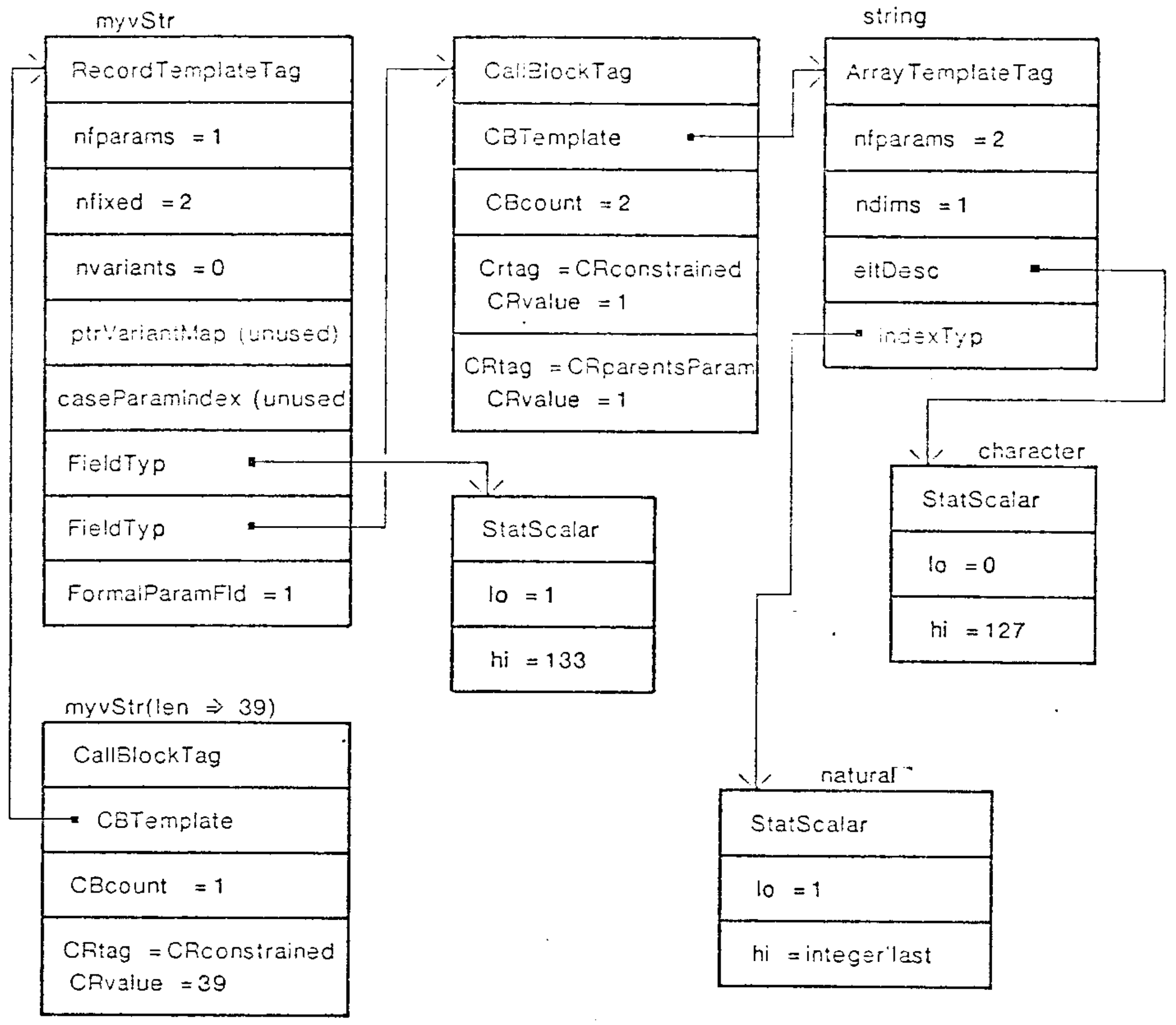
The descriptor for a record type is shown in Figure 2-5.

```
Tag: always RecordTemplateTag
------------------------------------------------
NFParams: number of formal type parameters to
     this type
------------------------------------------------
NFixed: number of fixed (i.e., non-variant)
     fields
------------------------------------------------
NVariants: number of variants (is number of
     when's). If zero, the next 2 fields,
     though present, are meaningless
------------------------------------------------
PtrVariantMap: pointer to another structure,
     that tells which case choices
     apply to a variant (not pictured)
------------------------------------------------
CaseParIndex: index of the formal parameter
     to use for constraining this case
------------------------------------------------
FieldTyp: pointer (absolute) to the type or
     subtype descriptor for this field
(Repeated for each field of the record)
------------------------------------------------
VariantTyp: pointer to a call block, which in
     turn points at a record type descriptor
     for this variant's component list
(Repeated for each variant)
------------------------------------------------
FormalParamFld: indicates which field of the
     record this formal parameter applies to
(Repeated for each formal parameter)
```

Figure 2-5: Record Type Descriptor

Figure 2-6 shows the record type template created for the record type `MyVStr` in Section 2.1.2. Also shown are the call block created by the `MyVStr(Len=>39)` discriminant constraint, and the array type template for the type `String`.

Variant records are handled by making each *variant component list* look like another record type descriptor. There is one variant component list per occurence of **when** in the declaration. The top-level record type descriptor then points at call blocks that point at these record descriptors. The call block is interposed to allow discriminants (parameters) of the outer record to be used inside a variant component list. For the variant record type `VR` in the example below,



```
type MyVStr(Len: Integer range 1..133 := 120) is
  record
    SVal: String(1..Len);
  end record;
```

Figure 2-6: Templates and Call Blocks for MyVStr and String

```
type VR(N: Integer range 1..4) is
  record
    case N of
      when 2..3 =>  M: Integer;
                    S: String(2..N);
      when 1 | 4 => T: Integer;
    end case;
  end record;
```

we get the type template structure shown in Figure 2-7.

In a variant record, we need a way of mapping a value of the discriminant (here, N) into an indication of which variant is appropriate for that value. We number the variants starting at one, and use these numbers to designate the variants. Let us call these designators *variant indices*. For the record type VR, the variant indices are "1" and "2", designating the "**when** 2..3" and the "**when** 1 | 4" variants respectively. The variant map is the structure that provides a mapping from the value of the variant record's **case** discriminant into the variant index appropriate for that value. This map is a table with one entry for each *choice* that occurred within the variants of this **case**; in the above example, the table has three entries, for "1", "4", and "2..3". The variant map table also a distinguished entry for "**others**", if it is present. The expressions used within choices are required by the language to be known at compile time. In nested variants, the same discriminant may potentially be used with more than one **case**; thus each **case** has its own variant map. The variant map is used during variable creation, including the construction of record aggregates, for it provides the mechanism to determine, from a value of the discriminant, which variant to create.

A record variable has two parts, fixed and dynamic. The fixed part is shown in Figure 2-8, and contains an offset to the base of the dynamic part. The dynamic part is shown in Figure 2-9. The storage for the dynamic part is allocated contiguously. The dynamic part is subdivided into fixed and dynamic parts for its fields. The fixed parts for all the fields come first. These have manifest sizes and offsets. Then come the dynamic parts for the fields. If a field has a dynamic part, then its fixed part contains an offset pointer to this dynamic part. Figure 2-10 shows the layout of a MyVStr record variable.

The component lists of variants are treated as sub-records. Any nested **case**'s become sub-records of the sub-records. While it would have been possible to *flatten* variant component lists up into the containing record variable, the sub-record approach reduces the amount of special-casing, since the scheme must handle nested records anyway.

When accessing a field in a variant, checking must occur to make sure that the current variant contains this field. This is easily done by comparing the variant index for that field (known at compile time) to the *CurVariantIndex* field of the record variable's representation.

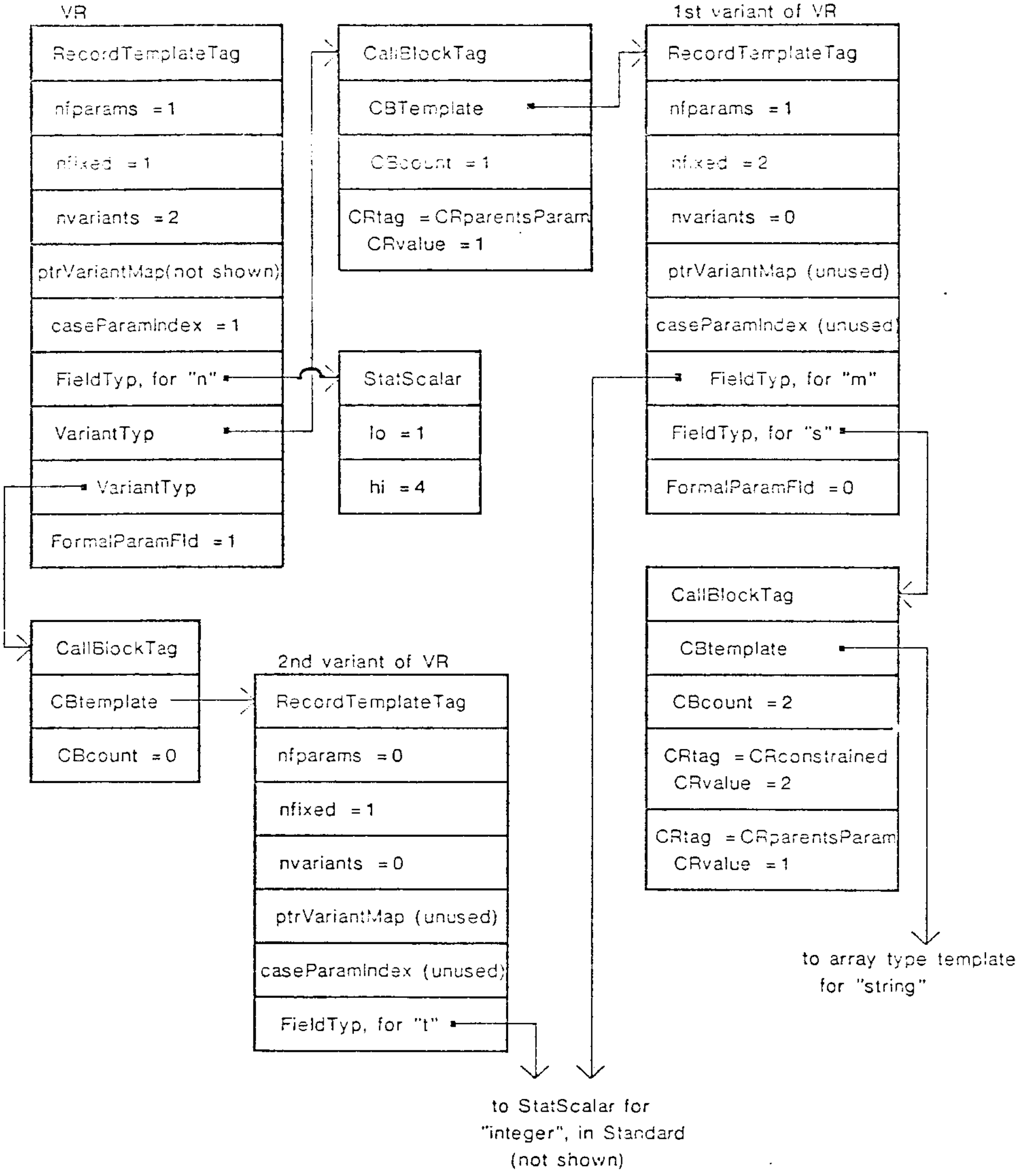


Figure 2-7: Templates for the Variant Record Type VR

```
Tag: always RecordVariableTag
---------------------------------------------
IsConstrained: true iff this record variable
     is constrained
---------------------------------------------
VarDynSize: total size of the dynamic part of
     this record variable.  Does NOT change
     on assignment
---------------------------------------------
RecOffset: offset to the dynamic part
```

Figure 2-8: Record Variable Fixed Part

```
Tag: always RecordDynVariableTag
---------------------------------------------
CurDynSize: current (i.e., in use) size of the
     dynamic part of the record's value.  May
     change on assignment
---------------------------------------------
CurVariantIndex: tells which variant is
     currently valid for this record
---------------------------------------------
First Field's Fixed Part
(Repeats for each field)
---------------------------------------------
First Field's Dynamic Part, if any
(Repeats for each field)
```

Figure 2-9: Record Variable Dynamic Part

2.2.6 Access Types

A variable of an access type consists of a fixed part that contains just the address of the referenced heap object; it has no dynamic part.

The type descriptor for an access type consists of a tag indicating that it is an access type, plus an absolute pointer to the accessed type or subtype.

Ada allows declaration of subtypes of access types; this permits supplying constraints on which heap objects the pointer may reference. The constraint on the access type means that only objects that satisfy the constraint may be referenced by the pointer. The compiler represents this subtype by keeping track of the address of the call block that was constructed during the elaboration of this subtype. This call block is used whenever a variable of this subtype appears in a context that requires constraint checking. The compiler uses both the variable and the call block, addressing them independently. For example, in passing a variable of an access subtype as an actual parameter, constraint checking may be necessary both before and after the call.

This implementation of access constraint checking is analogous to the implementation of subrange checking for scalar subtypes. The scalar variable and the scalar subtype descriptor are addressed independently, and the subtype descriptor (or, likewise, the call block) is shared amongst all instances of the subtype.

2.2.7 Variable Allocation and the Run-time Allocation Routine

Type templates and call blocks are used during the allocation of variables. A run-time routine interprets them to allocate the dynamic parts of variables. The dynamic storage is obtained by growing the stack. The routine also initialises the fixed parts of variables, filling in descriptor information. The representation of a procedure's stack frame thus resembles that of a record variable. For example, if the procedure has a local array variable, then its fixed part lies in the fixed part of the stack frame and contains offsets into the dynamic part of the stack.

The allocation of heap objects is handled by an alternative entry point into the allocator.

The run-time allocator could do default initialisation of variables if we were to augment record type descriptors by having each field point at a variable giving its default value. At this writing, our group has not yet decided how we wish to do such initialisation (see Section 6.4).

2.3 Support of Ada Operations

This section describes how several Ada operations are implemented.

2.3.1 Assignment

Consider assignment of records that contain dynamic arrays, such as the record type `MyVStr` in Section 2.1.2. After constraint checking has been done, assignments of `MyVStr` record variables may proceed by a block copy, using the size of the smaller variable (see Figure 2-10).

Now let us consider assignment of an array of arrays.

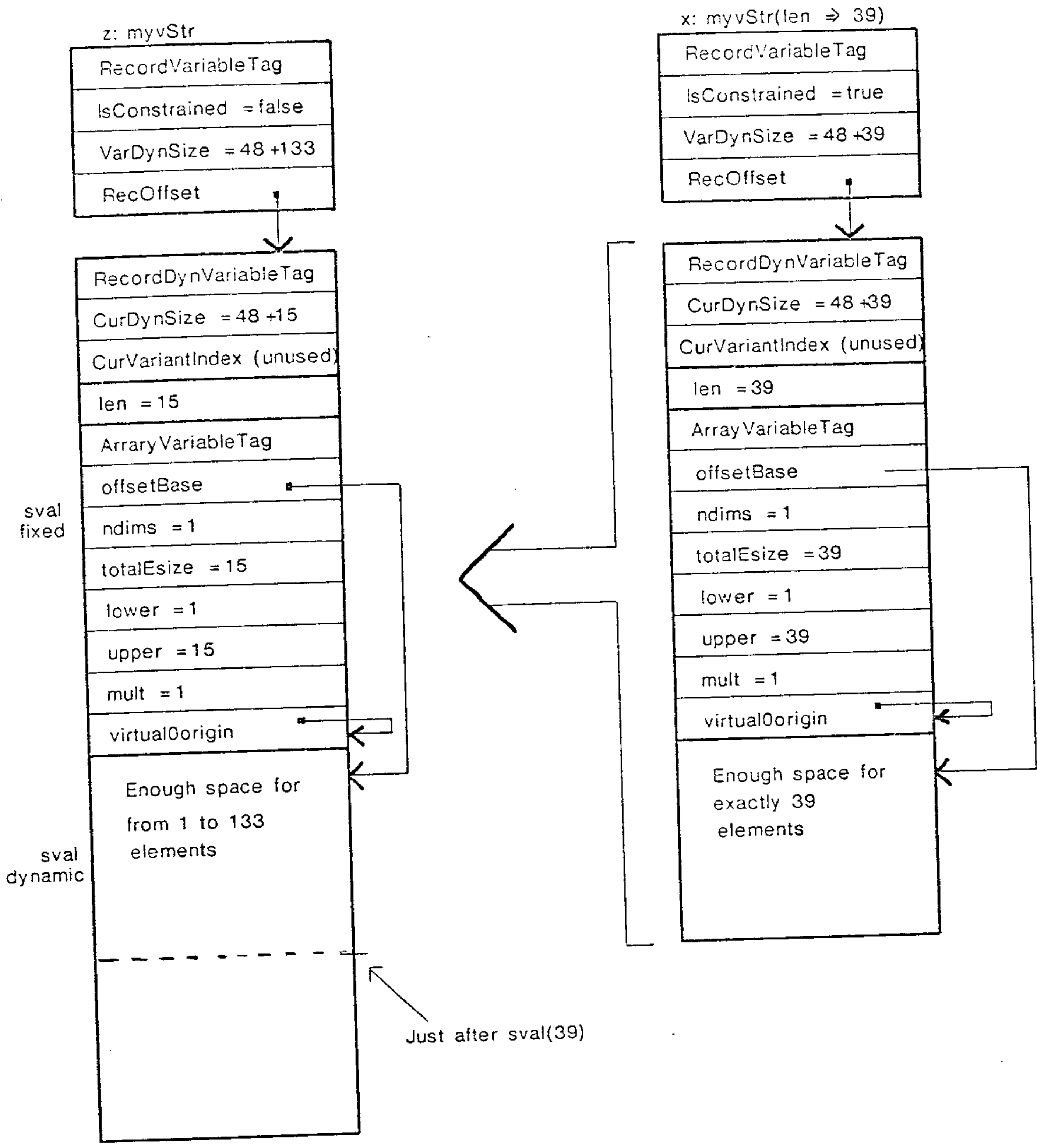


Figure 2-10: Assignment of the Constrained MyVStr Record X to the Unconstrained MyVStr Record Z

```
declare
    type TOuter is array (Integer range <>) of String(a..b);
    X: TOuter(1..10);
    Y: TOuter(11..20);
begin
    X := Y;
    ...
```

We represent a variable of such a type as it is written: the outer array variable is a one-dimensional array whose elements are array variables. This produces many inner array descriptors. It is a property of Ada that the bounds on all the inner arrays must be the same for two variables of the outer type, since these inner bounds must be given at the time of the outer type definition. For the variables to have different inner bounds, they would have to come from different type definitions and would thus be of different types. We can safely copy the array descriptors for the inner arrays over each other, as they all have the same representation. Therefore, the assignment X:=Y above may proceed by block copying the dynamic part of Y into the dynamic part of X. It would have been illegal to copy the fixed part of Y into X, since the bounds of X would then be wrong. If we had chosen to treat the array of arrays as a two-dimensional array, then it would obviously have been legitimate to block copy the dynamic part.

More remarks on array assignment, especially on constraint checking, are given in Section 4.3.1.

2.3.2 Aggregates

Aggregates are handled by treating them like temporary local variables. The constraints are those given in the aggregate. For example, a record aggregate of the record type MyVStr

```
(Len => 8, SVal => "Lovelace")
```

is treated like creating a local constrained record variable TempMyVStr8:

```
TempMyVStr8: MyVStr(Len => 8);
```

The run-time allocator is called to create the dynamic part on the stack and to fill in the fixed part. The values supplied in the aggregate are copied into TempMyVStr8 as in assignment.

The lifetime of the temporary variable TempMyVStr8 need be no longer than the statement that contains the aggregate. The lifetime problem for aggregates is analogous to that of function results whose size is not known at the time of the call, and thus may be approached in the same way. The function result problem and its solution are discussed in Section 4.4.1.

2.4 Design Issues and Alternative Implementations

2.4.1 Attempts at Sharing Descriptors for Variables

It is often apparent that several variables have the same constraints. Variables may have been declared together, or an explicit subtype name may have been introduced. It is interesting to consider storing the constraint information exactly once in a common descriptor, and sharing this descriptor among the variables. Such techniques have been suggested for the implementation of Euclid, both in the Euclid Report [20], and by Holt and Wortman [13]. The advantages to this approach are the space savings and, more importantly, possible reductions in variable initialisation complexity and overhead. However, the shared descriptor approach introduces complexities of its own. In passing an array variable to an unconstrained formal parameter, the shared descriptor must also be passed in order to provide the bounds. A more difficult problem is that the representation of a variable depends more on where it is allocated. For heap objects the constraint must be accessible via the access variable; this suggests either that the constraint will be stored with the heap object itself or that the access variable will be represented by two pointers. For dynamic arrays within records, each record variable will need its own copy of the array bounds, for the bounds are changeable on a per-record basis. The non-shared representations presented in this paper allow this kind of special casing to be avoided[1].

2.4.2 Representation Specifications

Our variable representations intertwine user-visible fields with user-invisible descriptor fields. It can be argued that this has a detrimental effect on the utility of Ada's representation specification facility. Our implementation does not support representation specifications, but, as a compromise, we could document the representations the compiler will pick. An alternative scheme that kept user-visible and user-invisible fields entirely seperate might allow users to comprehend more easily the compiler-generated layouts of the user-visible fields. For composite variables, two seperate composite structures would be built, one for each class of fields. Such a seperation would have its costs, especially for programs that have little dependence on the layouts. Overheads would be imposed on accessing a component of a variable, for two seperate address computation paths would be required. Procedure calls would require passing the invisible fields as an extra implicit parameter.

[1]Euclid has a mechanism like dynamic arrays within records, but it requires all instances of the record variable to be constrained. Thus the bounds of the array cannot change.

2.4.3 Block Comparison of Composite Variables[2]

Implementing equality and inequality for composite variables requires some strategy for coping with unallocated storage within the variable. A field-by-field comparison will of course work; it must be prepared to "walk down" arbitrarily many layers of composition. A block compare implementation must guarantee that the unused storage always contains an agreed upon value, such as zeros. This is slightly harder than it seems, for the amount of unallocated storage might change. Consider the program:

```
type Pair is record A,B: MyVStr; end record;
R,S: Pair;
X: MyVStr(Len => 39);
Y: MyVStr(Len => 15);
begin
...
R.A := Y;
S := R;
R.A := X;
R.A := Y;          -- Must be careful to zero unused
                   -- portions of R.A
if R = S then ...
```

When assigning a constrained variable Y to an unconstrained variable R.A, the unused space in R.A must be filled in with zeros. This implementation in effect shifts part of the cost of comparison onto assignment, with the obvious consequences for programs that contain many assignments and few comparisons.

Our representations can handle both the zero-fill and the field-by-field approaches. For the zero-fill, the *IsConstrained* bit in each record variable, as well as the *VarDynSize* field (which gives the size of the variable), would permit an assignment operation to discover that zeroing is required and to carry it out.

For the field-by-field compare, a general purpose comparison operation for records and arrays could be written as a run-time routine. This routine would be passed the two variables along with the type template. The type template would provide the comparison routine with the knowledge of the internal structure of the variables. The type template for a composite type points at the type templates for the components; this would enable the comparison routine to call itself recursively on the components. Constraint information is no problem, for it is present in every variable.

[2] This section draws on a report by Hilfinger [11].

3. The Front End

The Front End for Ada consists of two programs: a lexer/parser and a semantic analyzer/TCOL generator. The first was created from an LALR(1) parser generator and a grammar derived from the preliminary and revised language reference manuals [15, 17]. Both the parser and the semantic analyzer/TCOL generator are written in Simula.

For the rest of the chapter, the term *parser* refers to the program that implements the lexical and syntactic analysis phases, and the term *semantic analyzer* refers to the program that implements the semantic analysis and TCOL generation phases.

Section 3.1 discusses some semantic problems in Ada and how they were solved. Section 3.2 discusses the construction of semantic analyzers in Simula. An example illustrates the technique on a simple expression language and how it is used in our implementation. Section 7.2 presents some statistics on the working Front End.

3.1 Semantic Analysis Issues

The primary functions of the semantic analyzer are type checking, overload resolution, and name binding. This can be difficult in Ada, where the same name can refer to a number of different objects and the same syntactic structure can have a number of different meanings. The specific rules about the visibility, overloading, and use of identifiers are complex, changing subtly in different contexts. Situations where identifiers can be redeclared, hidden, and later made visible, complicate the symbol table facility.

3.1.1 Syntactic Generality

There can be more than one meaning to the same syntactic structure. For example, F(X) can have five possible interpretations:

1. Subprogram call. F is a subprogram; X is a parameter.
2. Entry call. F is an entry; X is an entry family index or formal parameter.
3. Array index. F is an array variable; X is an index.
4. Parameterless entry in accept statement. F is an entry; X is an entry family index (discrete range).
5. Conversion. F is a type name; X is an expression.

The grammar distinguishes statements (1,2) from expressions (1,3,4,5) and in (4) the F(X) comes after the reserved word **accept**. Further resolution requires looking up F in the symbol table and knowing more context. F(X) is initially determined to be a general INDEXED_NAME. When the meaning is determined, this piece of the derivation tree is replaced with a more specific entity such as ARRAY_ELEMENT.

3.1.2 Overload Resolution

Because of overloading, even when `F(X)` is known to be a procedure or entry call, we still do not know which `F` is denoted. The determination requires analysis of the parameter `X`, which might itself be a call. The preliminary language reference manual requires:

> The types, modes, and names of the parameters, together with the result type must be sufficient for the identification. The contextual information is propagated both ways, repeatedly until convergence [15].

It has been shown [8, 18] that four passes are sufficient for either a convergence or ambiguity to be recognised. Other results indicate that two passes are sufficient [22, 23, 29].

In trying to resolve a complex overload, it may not be necessary to analyze all parameters in detail. For example, consider the call `F(X,(3,5),Y);`. There may be several procedures named `F` that can take 3 arguments. Rather than trying to determine the array or record type of the aggregate `(3,5)`, it may be possible to determine `F` uniquely from the first and third parameters `X` and `Y`. Once this is done, the type of the second parameter is known. With this context information the analysis of the aggregate is much easier.

Our experience with the overload resolver indicates that the complexity is not in gathering the possible subprograms that could be used in a context or in the extra tree walks required for convergence. The complexity lies in determining the correct subprogram when the only distinguishing feature in the expression is the type of an aggregate. Ada's type equivalence rules are generally classified as name equivalence, but the processing to determine the type of an aggregate more closely resembles structural equivalence. Aggregates offer a very rich semantics with a minimal syntax. This results in large amounts of compiler code and execution time to determine the type of an aggregate.

Even if the user decides to avoid the complications of overload resolution by having unique names for all subprograms, the use of derived types causes implicit overloading of subprogram names. If the user additionally decides to avoid derived types, the predefined Standard package contains `Short_Integer`, `Long_Integer`, `Short_Float`, and `Long_Float`, which are derived. Derived types can also be inadvertently introduced. For example, neglecting to use the word `Integer` in

```
    X: Integer range 1..5;
```

is interpreted as deriving a new type from `Integer`.

3.1.3 Static Evaluation of Expressions

It is necessary that a construction such as

```
case X of
    when 1+1 => ...
    when 2 => ...
```

be recognised as illegal. This requires that the static expression 1+1 be evaluated. In addition to error detection, it is sometimes necessary to evaluate static expressions for type determination. For example, in deciding whether an aggregate can be of a particular type, the discriminants must be evaluated.

```
type R (X:Integer) is
    record
        case X is
            when 1 => null;
            when others => Y:Integer;
        end case;
    end record;
```

(1) or (3 , 3) could be aggregates of type R above, but (3) could not.

Static values for numeric literals are computed when the literal is recognised. Other expressions are flagged as static but are not calculated unless needed. This calculation is performed by a call to a recursive expression evaluation procedure.

3.1.4 Visibility of Identifiers

Ada, along with many data abstraction languages, provides the programmer many ways to control the visibility and use of identifiers in the program. Among the features of Ada that affect identifier visibility are:

- block structured scopes
- modules, i.e., packages and tasks
- renaming declarations
- overloaded identifiers
- private types
- forward declarations
- **use** clauses
- separate compilation

Symbol tables are used to collect identifiers for blocks, packages, tasks, subprograms, records, compound statements, and parameter lists. These symbol tables are linked together in the order they are to be searched.

Packages represent the first and probably most common departure from the Algol-like block structuring of identifier visibility. The implementation of symbol tables for packages must be able to handle the different scopes in a package specification, in a package body, and in a program that has opened a package with a **use** clause.

3.1.4.1 Symbol Tables for Packages

During the processing of a package specification or body, the symbol table appears to be connected in the same hierarchy as nested blocks. The problem is to permit the program inside of the package to reference all symbols in the specification and body but to restrict the program outside of the package to using only identifiers in the specification. Our implementation accomplishes the changing of identifier visibility (in a package) by using a linked list of symbols with two starting pointers. Consider the example below:

```
package Example is
        A: Integer;
        B: Integer;
end Example;

package body Example is
        C: Integer;
        D: Integer;
end Example;
```

A picture of the symbol table for this package is shown in Figure 3-1

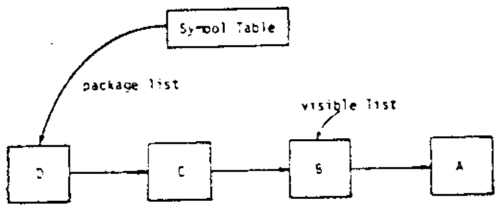


Figure 3-1: Symbol Table for a Simple Package

The symbol table lookup routine uses a single pointer that points to the beginning of the list of visible symbols. Package symbol tables keep two additional pointers: one to the beginning of the visible symbols for the package, one to the symbols visible outside of the package. When processing the body, the visible-symbol pointer is set to the entire list; outside the body, it is set to the middle of the list where the identifiers in the specification begin. Because the list is built backwards, the forward linear search finds only the permitted symbols.

3.1.4.2 Specifications and Bodies in the Symbol Table

Because packages, subprograms, and tasks can come in two parts, they are entered in a symbol table as two parts - one for the specification, one for the body. This is useful for verifying that respecifications of objects are consistent. Consider the example below:

```
package Example2 is
    procedure Init(X: in Integer := <exp>1);
end Example2;
.
.
package body Example2 is
    procedure Init(X: in Integer := <exp>2) is
        ....
end Example2;
```

The language permits $\langle exp \rangle_1$ and $\langle exp \rangle_2$ to be different texts as long as their expressions use the same entities in the same order.

3.1.4.3 Use Clauses

Use clauses represent an interesting change in the identifier visibility structure of the program. To a first approximation, the effect of a **use** clause is to copy the specified symbol table(s) and place them just before the symbol table for the current scope. The semantics of the **use** clause are more complex, and several additional steps must be taken. Three problems are present:

- No symbol in the current scope (or other open scopes) may be hidden by a symbol in the module specified in the **use** clause.
- Symbols in several **use** clauses may not hide each other. Should a conflict occur, both symbols must be removed from the duplicated symbol tables.
- The set of overloaded symbols within a package could be changed.

The first two problems are solved by checking that all duplicated symbols are not hiding other symbols before entering them into the symbol table.

The overloaded symbol problem is more subtle. Overloaded symbols are normally chained together in the symbol table, forming a list of permissible objects for an identifier. As a simple example, consider the program fragment:

```
declare
    ...
    procedure F(X:Integer) is begin null; end;
    procedure F(X:Boolean) is begin null; end;
    ...
begin
  null;
end;
```

The corresponding symbol table entries are shown in Figure 3-2.

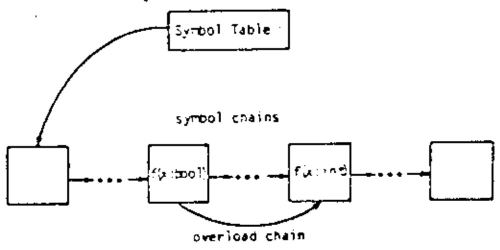


Figure 3-2: Symbol Table for Overloaded Identifiers

This works quite well should the relative positions of the overloaded symbols never change. **Use** clauses *can change* the available overloaded symbols. Consider the program fragment shown in Figure 3-3. Inside of the nested package, all three versions of procedure F are available because a package inherits all the symbols from its enclosing scope. Therefore the overload chain has all three procedures tied together. Blindly copying the symbols for package Inner to the beginning of the compound statement erroneously copies the overload chain leading from the F with a boolean parameter to the F with the character parameter. To avoid this, all overloading chains are broken when copying over a symbol table and reestablished in the new context.

3.1.5 Derived types

The inherited subprograms for a derived type can be represented as copies of the original subprograms. It may seem space inefficient to have multiple copies of predefined functions, such as = and /=, which differ only in the types of their parameters. Without multiple copies of functions, the symbol table access routines must be able to distinguish between predefined, declared, and derived procedures. We chose not to complicate the lookup procedures. Our strategy has been to stress

```
procedure Main is

    procedure F(X:Integer);
    package Outer is
        procedure F(X:Character);
        package Inner is
             procedure F(X:Boolean);
        end Inner;
    end Outer;

    procedure F(X:Integer) is begin null; end;
    package body Outer is
        procedure F(X:Character) is
           begin null; end;
        package body Inner is
             procedure F(X:Boolean) is
             begin null; end;
          -- all three expressions
          -- F(3), F(True) and F("A")
          -- are legal
        end Inner;
    end Outer;

begin
    declare
        use Outer.Inner;
    begin
    --  F(3) and F(True) are both legal
    --  F("A") is illegal
        null;
    end;
end Main;
```

Figure 3-3: Overloading and **use** Clauses

clarity and modularity and not efficiency. We have recoded critical modules to be more efficient after the algorithms and their implications were understood.

3.2 Implementation of the Semantic Analyser

This section describes the implementation of the semantic analyser. An understanding of Simula, and especially the *class* mechanism, would be beneficial. The relationship between Simula classes and Ada records is described in Appendix III.

We give several examples of *class skeletons*. A class skeleton is a class definition with only the parameters and visible components listed. In general, **boldface** is used to represent Simula keywords, *italics* to represent an English description of some omitted Simula code, and regular type face to indicate Simula code. Liberties are taken when translating grammar symbols into Simula identifiers. Classes are labeled by the identifier following the keyword **class**. If a class is defined as P **class** S, it is referenced as class S even though it is a subclass. P is termed the *parent class* or *prefix class* of S. In the descriptive text, SMALL CAPS are used for the names of classes. More details about Simula can be found in reference texts [2, 28].

3.2.1 Representing Syntactic Entities

The heart of the semantic analyzer is its use of Simula classes to represent grammar productions and Simula class instances to represent nodes in the derivation tree. Edges in the derivation tree are represented by Simula reference values, i.e., pointers to class instances. Non-punctuation terminal symbols are kept in class parameters.

The use of Simula to represent a derivation tree is illustrated with an Ada-like language of simple variables, numbers, records, record selection, and addition. A grammar for this language is given below:

```
<exp> ::= <name exp>
<exp> ::= <plus exp>
<exp> ::= number
<name exp> ::= <name exp> . identifier
<name exp> ::= identifier
<plus exp> ::= <exp> + <exp>

Start Symbol: <exp>
Nonterminals: <exp>, <name exp>, <plus exp>
Terminals: number, identifier, ., +
Punctuation: ., +
```

For each production A ⇒ XYZ, where A is a nonterminal and XYZ is a sequence of terminals and nonterminals, we write an empty class skeleton:

```
A class XYZ (X,Y,Z, ...);
begin
end;
```

Using this method for the sixth rule in the example grammar gives the class skeleton[3]:

[3]Only selected parts of the example are discussed. The final classes for each rule are given in Appendix II.

```
plus_exp class add_exp(left_exp,right_exp);
ref(exp) left_exp, right_exp;
begin
end;
```

Here the Simula identifier EXP corresponds to the nonterminal symbol ⟨exp⟩, the identifier PLUS_EXP to the nonterminal symbol ⟨plus exp⟩, the identifier ADD_EXP names the right hand side of the 6th production. The punctuation symbol + is discarded.

There is also a class to represent the root of the derivation tree. Because the root isn't derived, there is no parent (superclass) in its class definition. The class skeleton for the start symbol ⟨exp⟩ is:

```
class exp;
begin
end;
```

This class is defined as a parent class for rules that have the start symbol as their left hand sides.

Edges in the derivation tree are references (pointers) to class instances. These references are held in the class parameters and represent the derivation rule that reduced the right hand side of the production. For the class PLUS_EXP, the two parameters reflect the rules that were used to generate the two right hand side expressions.

In the general case, each derivation in a parse produces a reference value to be stored in a class parameter. Not all of these derivations are useful for semantic analysis. Extra references are eliminated by the use of the subclass facility. When the right hand side of a production is a single nonterminal symbol, the parameter list is omitted. For example, the class for the first rule is:

```
exp class name_exp;
begin
end;
```

Rules with ⟨name exp⟩ on the left hand side use NAME_EXP as a parent class. Derivations that include the reduction ⟨exp⟩ $\Rightarrow$ ⟨name exp⟩ do not create a reference from class NAME_EXP to the class that is used for the next rule. Subclasses of NAME_EXP are automatically subclasses of ⟨EXP⟩ so that an instance of a subclass of NAME_EXP describes the application of two rules rather than one.

An example derivation tree is given in Figure 3-4. The figure illustrates the expression `R.S + 3`. Boxes in the figure represent class instances of the class named at the bottom of the box. Lines between boxes represent references to class instances held in parameters. Note the use of subclasses and references to classes to represent edges in the derivation tree.

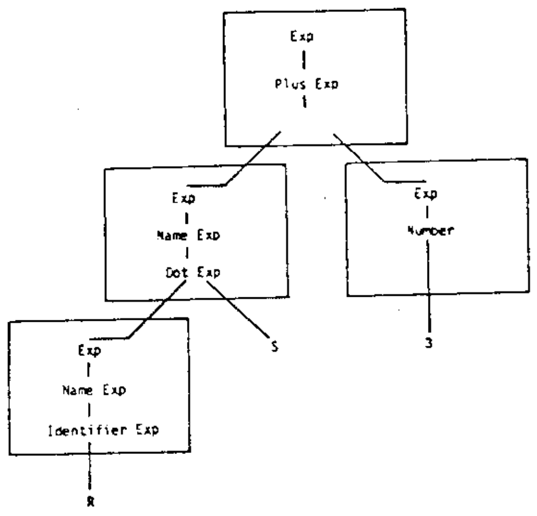


Figure 3-4: Derivation Tree for R.S + 3

3.2.2 Operation of the Semantic Analysis Phase

Semantic checking is performed by walking the derivation tree and verifying that each production is consistent with language semantics. The very nature of context free languages prohibits any prior knowledge about the productions applied after a given rule is used. Therefore, a node cannot explicitly name the correct procedure for nodes below it without examining its children. This requires placing information about possible derivation sequences in each class and distributes semantic knowledge widely across the entire program. Instead, we chose to confine all information about the semantic checking for a production inside the class for that production. This was accomplished by the use of *virtual* procedures in Simula.

Each class has a *semantic check* component which is a virtual procedure. This procedure encodes the language semantics for that class's production. A node initiates semantic analysis of its subtrees by calls on the SEMANTIC_CHECK procedures of its subtrees. Simula guarantees that the procedure of the instantiated class is executed.

Other properties of a nonterminal or production are represented as other class components. For example, the classes that represent the processing of expressions have a *type* component. In our small sample language, there are only three types: integer, real and record. These are represented as integers, so the class component for a type is an integer variable.

Recall the program fragment `R.S + 3` and the derivation tree in Figure 3-4. Assume that the expression appears on the right hand side of an assignment statement. The semantic check in the assignment statement analyzes the left hand side of the statement and determines the necessary type for the right hand side. This type is passed to the SEMANTIC_CHECK function of the right hand side. Since the right hand side of an assignment statement can be any expression, it is represented by an object of class EXP. The use of virtual procedures causes the correct semantic processing to be done for any subclass of EXP. In the example, the SEMANTIC_CHECK function for ADD_EXP objects would be invoked, which in turn calls the necessary semantic checks for its constituents. If the expression is semantically correct, all the SEMANTIC_CHECK calls return `True` and the right hand side of the assignment statement has been processed.

This technique allows for a very modular design with great flexibility. Adding another type of expression is trivial. One merely adds another subclass of EXP and the semantic analysis for this expression. The call to check it semantically is exactly the same. The meaning of a particular construction is also well localised. For example, the requirement that the types of the operands for addition must be of the same numeric type, is manifested *only* in the class that handles addition expressions.

3.2.3 Limitations of the Class/Virtual Procedure Technique

Creating classes for *every* production results in a tremendous number of useless classes and duplicate attribute code. For example, many languages, including Ada, have a list of productions for describing arithmetic expressions. One reduces EXPRESSION ⇒ RELATION ⇒ SIMPLE EXPRESSION ⇒ TERM ⇒ FACTOR ⇒ PRIMARY ⇒ LITERAL ⇒ NUMBER ⇒ INTEGER NUMBER to derive a simple integer. Eight subclasses are generated when only one is necessary. Arithmetic expressions also cause duplicate SEMANTIC_CHECK code. Although there are two productions for addition and subtraction, there is little difference in the processing of these expressions. The subclasses generated for the two productions have the same code for semantic checks. Further, if a change is required in the semantics of arithmetic expressions, the change would be required in both subclasses. This strays from a modular breakdown of the language semantics.

Our solution is to collapse subclasses when their semantics are sufficiently close. In the implemented system, LITERAL_EXPRESSION is an immediate subclass of EXPRESSION without any intervening subclasses. These decisions are purely subjective. There are cases where we split a class that embodied too many attributes, and cases where we combined two classes that did the same processing. A table of the class structure as currently implemented is contained in Appendix I.

3.2.3.1 Private Types and Forward Declarations

Private types and forward declarations present unique problems in our implementation of the front end because of Simula's lack of record assignment, or more technically, the inability to use "node-overwrite" techniques for updating elements in list structures[4]. When a specification is encountered in the program, a symbol is placed into the symbol table as a placeholder for the complete declaration. It is quite likely that other pieces of the program will acquire a reference value for the placeholder's class instance. This prohibits the discarding of the temporary place holder and replacing it with the actual declaration once the complete definition has been processed. To circumvent this problem, an indirect object, a "mimic" which can take on the appearance of another type, is used as a placeholder. References to the actual object are passed on to the actual symbol by means of virtual procedures. If for some reason a request is made for some attribute of the complete object before it is defined, the placeholder issues an appropriate diagnostic message.

[4]Standard texts discuss node-overwrite vs. pointer-swing techniques for manipulating data structures [32].

4. The TYPEREP and XFORM Phases

In this chapter we explore some aspects of the translation of preliminary Ada into MIL. We give only glimpses of the total translation process and many aspects of the language will be ignored. Features that we will *not* discuss include: access, float and fixed types, tasks and generic definitions and instantiations.

4.1 An Overview of MIL

MIL is a low-level system implementation language in the spirit of Bliss [30]. It is described in detail in Appendix VI.

MIL was designed as a tool for language translation and not as a general purpose programming language. In fact, there is no source language representation for MIL; it exists only in a graph form (LGN, see Section 1.3.1). In this chapter we will present MIL in a synthetic source-like form.

Some of the cogent characteristics of MIL are:

- It is expression oriented. Most statements return a value and an expression can always be used as a statement.
- It is a typed language. The types are *integer* and *location* (pointer).
- Explicit dereferencing is required in all contexts. For a variable with name `X`, the "address of `X`" is denoted by `X` and its contents by `@X`. `@` is a general purpose dereferencing operator.
- Variables are fixed in size at compile time. There is no primitive dynamic allocation mechanism.
- The only data structuring mechanism is similar to a Pascal record.

Throughout this chapter we will show how various Ada constructs are encoded in MIL.

The heart of MIL is its typing/structuring mechanism. Integer (**int**) and location (**loc**) are the only types available. An **int** may be qualified with a *range*, as in

```
int 0..127;
```

The code generator may use the range to optimise storage allocation.

Actual storage in MIL is described by the use of *descriptors*. A MIL descriptor (**desc**) is something like a Pascal record, except that the fields are denoted positionally, rather than named. For instance,

```
desc(int, loc, int 0..127 := @C+3)
```

Initialisation expressions (i.e., `:= @C+3`) may appear in a **desc** and are evaluated by the code generator and assigned to the corresponding fields at run time.

Descriptors are used to describe storage in a machine-independent manner, both for allocation and access. For example, variables in MIL are declared by associating a descriptor with them:

```
Tuple : desc(int, loc);          ! Define a descriptor
X : Tuple;                       ! Declare a variable
```

or, equivalently

```
X : (int, loc);                  ! Implicit descriptor definition
```

This would cause storage to be allocated for an integer followed by storage for a location. One would access the first and second fields of X by

```
access(X, 1, Tuple)
access(X, 2, Tuple)
```

respectively. The MIL **access** construct requires an explicit descriptor since the object being accessed may be the result of an expression and have no associated descriptor.

The usual complement of scalar operators is provided, including: the standard arithmetic, relational and assignment operations of **int**s; equality, inequality and assignment of **loc**s; addition of an **int** and a **loc** returning a **loc** (used in offset calculation, see Section 4.3.1); coercion operators from **int** to **loc**, and vice versa.

The primitive constructs provided in MIL for manipulating composite objects are simple. Only simple *block copy* and *block compare* operations are available. Manipulation of composite objects is done by providing explicit instructions to perform the desired action.

The statement structuring facilities include **routines** and **blocks**. A MIL **block** or **routine** may contain local declarations, a body, and an exception handler. A **routine** is recursive and may have a scalar return value. A formal parameter must have a descriptor that consists of a single field, thus restricting formals to be scalar. Correspondingly, all actual parameters in a routine invocation are expressions, and hence yield scalar values. Parameter passing is by value; other mechanisms must be implemented within this framework (see Section 4.3.2.1).

A **block** also contains a tag to control the lifetime of dynamic objects (see Section 4.4.1). This tag tells the code generator that at block exit it must restore the stack to its state prior to block entry. This allows XFORM to control the lifetime of dynamic objects by the placement of deallocation indicators.

4.2 Translating Declarations

MIL **blocks** and **routines** allow local MIL declarations which are used in the translation of Ada declarations. Because declaration elaboration in Ada can involve computing complex expressions, MIL allows statements to be interspersed with declarations.

4.2.1 Type Declarations

All types except for scalar types are represented at run time. It is an important consequence of our type template design that the structure and contents of type descriptors are known at translation time. This is a fine point, and is due to the Ada language definition. The reader may be tempted to claim that the **type** declaration

```
type T is array(N..M) of Integer;
```

defines a type that requires template information known only at run time. However, the language definition states that this declaration generates two implicit declarations: an anonymous **type** declaration of the form

```
type anonymous is array(Integer range <>) of Integer;
```

and a **subtype** declaration

```
subtype T is anonymous(N..M);
```

The Front End performs this transformation and XFORM sees only canonical type and subtype definitions.

Implicit **type** amd **subtype** declarations are created in many contexts in Ada. For example, variable declarations may construct a subtype explicitly[5]:

```
X : record Left, Right : Integer; end record;
```

As a less obvious example, **for** loops generate both a **type** and a **subtype** declaration for the range of the index variable.

Upon encountering a **type** definition, XFORM creates a MIL variable with a descriptor whose structure corresponds to that of the template for this type. The variable descriptor also contains initialisation expressions that assign the required type information. This representation is satisfactory since the structure of the template is known and contains only a fixed part.

[5]This declaration is legal in preliminary Ada but illegal in revised Ada.

As an example of this translation, consider the following declarations

```
    subtype Elt is <some definition>;

    type Elt_Array is array(Natural range <>) of Elt;
```

The MIL translation of this is shown in Figure 4-1.

```
! *******************************
! MIL descriptor and template for  Elt

Elt_desc : desc(<descriptor for Elt>);
Elt_template : Elt_desc;

! **********************************
! MIL descriptor and template for
! type Elt_Array is array(Natural) of Elt;

Elt_Array_desc : desc(int := 3,        ! Tag indicating an array type template
                      int := 2,        ! # "parameters" to this template
                      int := 2,        ! # dimensions
                      loc := Elt_template);
                                       ! Pointer to component subtype descriptor

Elt_Array_template : Elt_Array_desc;
```

Figure 4-1: Example of MIL for Type Descriptors

Notice the initialisation of the fourth field of `Elt_Array_desc` to the *address* of the subtype descriptor for `Elt`.

The MIL statements in Figure 4-1 are attached to the local declarations of the smallest enclosing unit at the point of the declaration. The translation of some Ada constructs (e.g., **for** loops) cause a local **block** to be created expressly for the purpose of containing non-user-generated type and subtype definitions.

4.2.2 Subtype Declarations

Subtype declarations are handled similarly to **type** declarations in that the basic action is to create a MIL variable and attach it to the local declarations of the enclosing unit. However, **subtype** declarations may contain execution-time expressions. Extending the example in Figure 4-1, the MIL translation of the declaration

```
subtype EA is Elt_Array(A+B..C);
```

is shown in Figure 4-2.

```
! ************************
! MIL descriptor for
! subtype EA is Elt_Array(A + B..C);

EA_desc : desc(int := 4,          ! Tag indicating call block
               loc := Elt_Array_template,
                                  ! Pointer to type template
               int := 2,          ! # "actual parameters"

              ! The next two fields are the actual parameters.
              ! Note that they are initialised with run-time expressions.

               int := @A+@B,
               int := @C);

! ****************************
! Descriptor (call block) for EA

EA_call_block : EA_desc;
```

Figure 4-2: Example of MIL for Call Blocks

4.2.3 Variable Declarations

Translating a scalar variable declaration is straightforward since scalars have only fixed parts. A scalar is represented by a MIL variable with a descriptor of one field whose type is int and whose range is defined as appropriate. For instance, the Ada variable

```
    Life : (Static, Local, Global, Register);
```

is translated into the MIL variable

```
    Life : (int 0..3);
```

Composite variables contain a dynamic part which must be allocated via a call to the run-time storage allocator. For the Ada declaration

```
    A : EA;          -- EA as in Figure 4-2
```

the process is indicated in Figure 4-3.

```
    ! **************
    ! Translation of
    !  A : EA;

    A_fixed : (<descriptor for array dope vector>);

    call alloc(A_fixed, EA_call_block);
```

Figure 4-3: Allocation of a Composite Variable

The storage allocator, *alloc*, is called with the address of the fixed part of A and the address of the call block for EA. The allocator will obtain and initialise the storage for A on the dynamic part of the run-time stack and initialise the fixed part. The dope vector for A will have its bounds and multipliers filled in and its offsets will be made to refer to the dynamic part.

4.2.4 Subprogram Declarations

Ada subprogram declarations require translation into executable code for two reasons: (i) default parameter values must be evaluated, and (ii) the subtypes of formal parameters, and any return value if a **function**, must be elaborated.

The Front End transforms all constructs dealing with default parameters into a canonical form. The details of the transformation are unimportant; it enables XFORM to work without knowing about default parameters.

Elaboration of formal parameter and return value subtypes presents no difficulties in general. Their subtypes are translated as any other and the MIL declarations/statements become part of the local declarations of the unit enclosing the subprogram.

However, unconstrained array and record subtypes may occur as formal parameter and return value subtypes. Unconstrained subtypes are easy because they require no translation. An unconstrained non-scalar formal parameter will obtain its constraint information from an actual parameter and needs no run-time type descriptor. For return values, the subtype is only needed at compile time for type checking at the call site and so we dispense with a run-time subtype descriptor. Since no constraint values appear in an unconstrained subtype, no expressions need to be elaborated.

4.3 Translating Statements

Many Ada statements translate directly into MIL. Scalar assignments and **if**, **for**, and **while** statements all have explicit MIL equivalents, for example. There are statements that require some effort to implement correctly.

4.3.1 Array Assignment

The semantics of array assignment require that the number of elements in each dimension of the source and destination arrays be equal. If the array lengths are unequal an exception is to be raised. Preliminary Ada does not specify the appropriate exception. In revised Ada `LENGTH_ERROR` is provided for this purpose.

For singly-dimensioned arrays the test for equal lengths can be performed by comparing the size fields stored in the fixed part of the array object (Section 2.2.4). This is true because our context-independent implementation scheme guarantees that two arrays of the same type and same number of elements will be represented identically. For multi-dimensional arrays the lengths of each dimension must also be compared. This is only slightly more complicated since the dope vector bounds multipliers gives this information almost directly.

Copying the array value can be done by a block copy operation since the array dynamic part, which contains the value, is always allocated contiguously and contains no absolute pointers to subobjects. MIL provides a **copy** statement for just such a purpose, the syntax of which is

copy(*<source location>*, *<destination location>*, *<size>*)

For array assignment, the *source* and *destination locations*, which are the bases of the dynamic parts, are computed using the offsets stored in the dope vectors. The location of the value of array `A`, for example, is computed by

```
A_desc  :  desc(<descriptor for array dope vector>);

...@access(A,2,A_desc) + access(A,2,A_desc)...
```

The addition expression computes the base location by adding the contents of the second field of the dope vector to its address. This field was initialised as an offset from its own location by the storage allocator at object-creation time.

The size of an array is found by accessing the size field within its fixed part. Since the length check has already been performed the arrays have the same size and either array may be used for this computation.

4.3.2 Subprogram Invocation

Subprogram invocation translation is a complex issue. We will not attempt a detailed analysis but give only an outline of the implementation. The primary sources of complication are parameter passing and return values. The former problem is dealt with in the next section and the latter within the section on expression translation (Section 4.4.1).

4.3.2.1 Parameter Passing

The translation of parameter passing is complicated by imprecise semantics in preliminary Ada. There is much question about the desired effect of exceptions on **out** and **in out** parameters and the legality of modifying the constraints of actual parameters. Lacking a full definition we have chosen an implementation that (i) is relatively easy to implement, and (ii) provides "reasonable" semantics. In revised Ada the semantics has been made precise and corresponds to the semantics of our implementation.

Parameter passing is done with a call-by-reference mechanism in which formal parameters are represented by MIL variables whose associated descriptors are

```
desc(loc)
```

This provides a variable with a single pointer as its representation. All parameter accesses are indirect through the MIL formal. The indirection is indicated explicitly in MIL which provides only call-by-value semantics.

An array or record actual parameter is passed by sending the location of its fixed part. This implies that all checking on the value occurs according to the actual subtype, not the formal. If an exception causes execution to leave the subprogram **out** and **in out** parameters will have been modified.

Passing scalar parameters is more complex due to our special casing of storage for scalar objects.

Consider an `Integer` formal parameter whose value requires four bytes of storage. The code generator has no information as to the size of an actual parameter, which may occupy one, two or four bytes on the VAX. The size of the actual parameter must be known to enable proper manipulation of its value. The following scheme allows a uniform reference to scalar actual parameters.

For each scalar actual parameter a temporary MIL variable is created whose MIL descriptor is identical to that of its corresponding formal parameter. If the binding is **in** or **in out**, the actual parameter is assigned to this temporary with appropriate range checking. The address of the temporary is then passed to the subprogram. This guarantees that all actual values are represented identically to the formal parameter subtype. Upon return from the subprogram, all **out** and **in out** actual parameters are assigned the value contained in their respective temporaries and appropriate range checking is performed.

Constraint checking on scalars is done only at subprogram entry and exit. If an exception is raised within the subprogram body, scalar **out** and **in out** parameters will not have been modified.

4.4 Translating Expressions

As with statements, many Ada expressions have obvious MIL equivalents. We discuss some of the more interesting expression translations in this section.

4.4.1 Function Return Values

Manipulating array and record function return values is a difficult task as there is no convenient place to allocate the return object. If it is allocated in the called routine's context then normal function return will deallocate the stack frame and destroy the object. Storage for the object could be placed in the caller's context if its size were known prior to invoking the function. The possibility of unconstrained return values make this impossible since the size of such an object may not be known until the function returns.

We have decided to use a *hole in the stack* implementation which operates by leaving the return object in the called routine's stack frame. The function returns in a special manner that leaves the stack frame intact, and the object untouched. Normally the storage for the function stack frame would remain until the stack frame "above" it was exited. This is unacceptable due to our run-time machine model which utilises a single stack for expression evaluation and for dynamic allocation of local variables (see Chapter 5). The run-time stack will become inconsistent if a return object "lives" for too long.

The mechanism for controlling object lifetimes in MIL is the block deallocation tag mentioned in Section 4.1. This forces XFORM to be responsible for controlling the lifetime of return objects. Obtaining the proper lifetime is tricky. Consider the following Ada statements, where A is an array, F is an array-valued function and K and N are Integers.

```
A := F(K);
N := N + F(K)'FIRST;        -- Legal only in revised Ada
```

In the first assignment statement the lifetime of the return value of F is the entire statement. The second statement requires deallocation of the value after the 'FIRST inquiry is complete. XFORM operates by knowing in any particular context whether it is possible for the lifetime of a return object to extend beyond the context.

A subprogram call must be treated carefully when one of its arguments is a function call having an unconstrained return value. Consider the following call, where G is a procedure taking an array and an integer as parameters.

```
G(F(K),N)
```

The translation of this is sketched in Figure 4-4. The storage for the inner MIL block is not deallocated when the block is exited, for this storage contains the array object that is being passed to the procedure G. The deallocation tag on the outer MIL block will cause the stack storage for both the inner and the outer blocks to be reclaimed once the outer block is exited.

```
BeginMILblock, with deallocation_tag = yes
    temp_loc := BeginMILblock, with deallocation_tag = no
                        Call F(K), leaving its frame allocated,
                        and yield the address of the returned array
                        as the value of this block;
                EndMILblock;
    temp_integer := N;
    G(@temp_loc, temp_integer);
EndMILblock;
```

Figure 4-4: Translation of G(F(5),N)

4.4.2 Subscripting

MIL has a **subscript** expression which renders the translation of Ada subscripting trivial. A primitive operation of this power may seem out of place in such a low-level language. It was included in MIL because the VAX provides a machine operation, the INDEX instruction, which allows extremely efficient machine language to be generated for the Ada subscript operation.

4.4.2.1 Slicing

Slicing an array produces a new access path but not a new object. This new variable has the same type as the sliced array but its subtype constraints are the expressions appearing in the slice.

To represent the new variable XFORM creates a MIL variable to represent the array's dope vector. A call block is allocated for the new array subtype but the actual bounds values cannot be filled in until the slice expression is evaluated. MIL statements are generated to elaborate the integer expressions in the slice and to assign the appropriate constraint values to the call block.

The dynamic part of the new variable starts at the element of the sliced array selected by the first integer expression in the slice. (E.g., the first component of `A(N..M)` is `A(N)`.) This location is computed with a **subscript** expression. Next the fixed part of the new array variable is set to refer to the computed dynamic part. This is identical to the function performed by the storage allocator, except that the storage allocator also creates a dynamic part. There is a second entry point into the allocator that performs exactly the desired actions. After the call to this entry point, the MIL variable is the result of the slice.

5. The CODE Phase

This chapter describes CODE, the final code production phase of the Charrette compiler. This phase is concerned with the impact of the VAX architecture and Unix[6] operating system on the code generation process.

5.1 Machine Model

No compiler system treats the full hardware available as a monolithic résource. Instead, some basic run-time model for the architecture is adopted. Our abstract target machine is a stack machine, both because the VAX supports a stack environment with its addressing modes, and because it is a reasonable model applicable to many machines. This choice impacts

- Instruction set utilisation
- Expression evaluation
- Subprogram calling conventions
- Address space segmentation

The implementation of these abstract stack machine operations with VAX instructions is discussed below. When possible, the philosophy, architecture, and conventions of the VAX system as published in the VAX manuals [5, 6, 7] are followed.

5.1.1 Hardware Utilisation

There are several sizes of objects on the VAX. *VAX objects* are defined as longwords (64 bits), words (32 bits), halfwords (16 bits), and bytes (8 bits). This contrasts with *Ada objects* which include arrays, records, tasks, and scalars. To simplify operations, every stack objects is a longword (4 bytes). Conversion to the correct size is performed when a datum is transfered to or from the stack by the CVT instructions, of which the MOV instruction is a special case with no conversions performed. When no conversion is necessary, it is possible to use the more efficient PUSH instruction. Technically, VAX instructions contain a designator, such as L, W, H, or B, which indicates the size of the operands. For simplicity, we omit these designators and assume that stack operands are always a longword in size.

The VAX supports most stack operations in its addressing modes and register allocation. By convention, register 14 is the stack pointer and like the PDP-11, the stack grows downward. Pushing VAX objects on the stack is accomplished by using a MOV instruction with the autodecrement addressing mode on the predefined stack pointer. Popping the stack is accomplished by using the autoincrement addressing mode in a MOV instruction.

[6]We use the Berkeley Paging Unix system, with local modifications for exception handling.

5.1.2 Expression Stack

Arithmetic expressions are implemented with the autoincrement/autodecrement modes in three address instructions. For example, addition is carried out by

```
ADD3 (SP)+,(SP)+,-(SP)
```

which pops the first two operands off the stack, adds them, and pushes the result back onto the stack[7]. Assignment is done by indirection through the top of the stack. For example, the Ada scalar assignment statement

```
X := Y;
```

is implemented by the series of instructions:

```
PUSHA X             ; push address of X
PUSH  Y             ; push value of Y
MOV   (SP)+,@(SP)+  ; place value at address
```

After pushing the address of X and the value of Y, the stack is popped twice and the value is stored at the address.

5.1.3 Call Stack

A stack machine does all of its calculations on the stack. Conceptually, a subprogram call is just another expression calculation, albeit one with side effects. Activation records for subprogram calls are allocated on the same stack as expressions. The actual linkage is shown in Figure 5-1 and is partially generated by the CALLS instruction.

The VAX architecture predefines several registers for maintaining the context of a subprogram call. These conventions are used and extended as follows:

PC	*Program Counter*, register 15. This holds the address of the next instruction to be executed.
SP	*Stack Pointer*, register 14. This points to the last byte on the stack.
FP	*Frame Pointer*, register 13. This points to the beginning of the current activation record.
AP	*Argument Pointer*, register 12. This points to a block of storage that contains the actual parameters passed to the current subprogram.

[7]There are some optimisation routines in the code generator that recognise this situation and can replace it with the more efficient ADD2 (SP) + ,(SP) instruction. Since these optimisations are a departure from the stack machine and are not always applicable, they will not be discussed here.

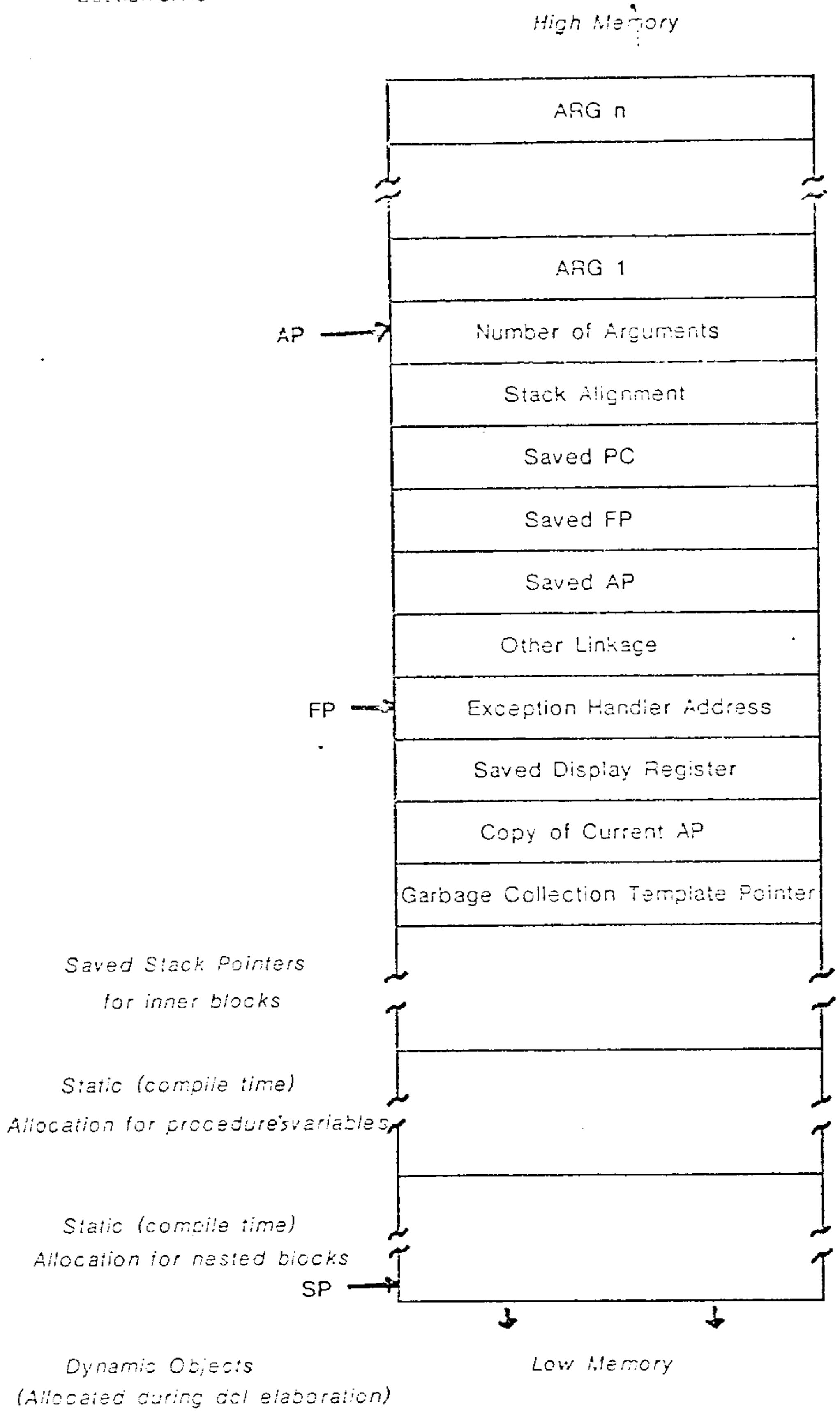

High Memory
ARG n
ARG 1
AP
Number of Arguments
Stack Alignment
Saved PC
Saved FP
Saved AP
Other Linkage
FP
Exception Handler Address
Saved Display Register
Copy of Current AP
Garbage Collection Template Pointer
Saved Stack Pointers for inner blocks
Static (compile time) Allocation for procedure's variables
Static (compile time) Allocation for nested blocks
SP
Dynamic Objects (Allocated during dcl elaboration)
Low Memory

Figure 5-1: Activation Record for a Subprogram

TP *Task Pointer*, register 11. This points to the current task control block for the executing thread of control. This register is the only one whose use is not defined in the architecture, but by the run-time system.

The first four registers (all but TP) are saved, restored, and loaded automatically by the CALLS and RET instructions. The TP is manipulated by the run-time system and normally is unchanged between subprogram calls.

In keeping with the stack machine philosophy, subprograms have their parameters pushed on the stack, and are invoked by a CALLS instruction. The context is saved (old PC, FP, AP) on the stack and replaced by the state of the current call. Allocation of local variables and enabling of an exception handler is performed by code at the beginning of each routine. The RET instruction, which automatically pops the stack and restores the state registers, is used to return from a subprogram.

A conventional scheme is used to implement block structured name scopes [9]. Storage allocation for all packages and blocks is raised to the enclosing subprogram level. Disjoint blocks in a subprogram share storage for their variables. Storage for *static* (size known at compile time) parts of Ada objects is allocated on the stack when a subprogram is entered[8]. Storage for Ada objects that have a compile-time-undetermined (*dynamic*) size is done during declaration elaboration. Storage is released on block exit by resetting the stack pointer to its value before the block was entered. Storage for subprograms is automatically released by the RET instruction.

5.1.4 Address Space Segmentation

We partition the address space as shown in Figure 5-2. The stack grows contiguously from the high end of memory. The lowest part of the address space is filled with the machine language for the program and run-time routines. The heap grows from the end of the program towards the stack.

5.2 Difficulties with Implementing Subprograms

There are several interactions between the simple stack model and Ada's semantics that cause the implementation to deviate from VAX conventions. These include:

- Uplevel references of variables
- Uplevel references of parameters
- Returning Ada objects from functions
- Sizes of parameters
- Heaps and secondary stacks

Each of these problems is discussed, along with the implemented solution.

[8]The layout of activation records is determined at compile time by the ACTREP phase.

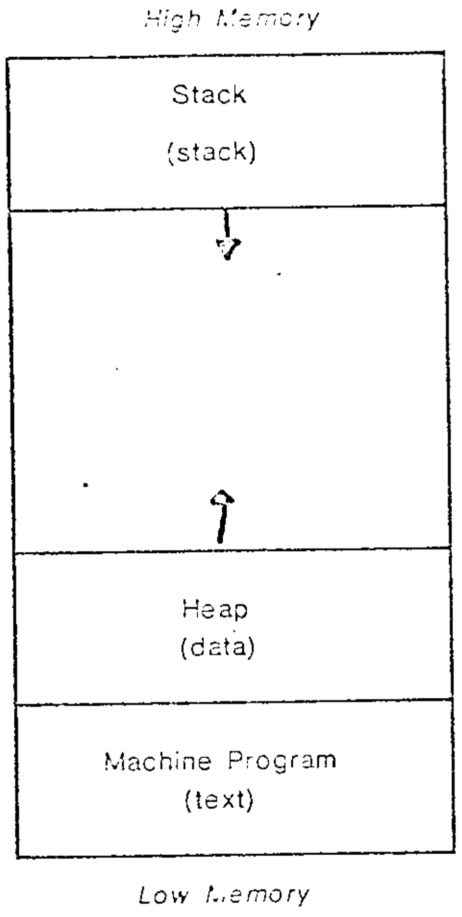


Figure 5-2: Partitioning of Address Space for Execution Objects

5.2.1 Uplevel References of Variables

A display is provided for referencing variables that are not in the current activation record. Display registers are kept in the task control block and are referenced through the TP register. Variables in the current subprogram are referenced through the FP register, allowing easy access. A variable that is stored *n* bytes from the beginning of the current activation record is located at VAX address -n(FP), an operand that can be used directly in VAX instructions. Address calculation is not as easy for references to variables in outer subprograms. Four steps are required:

- Get address of task control block
- Add in offset for correct display register
- Get contents of display register
- Add in offset for variable

This translates into three VAX instructions:

```
PUSH    i(TP)           ; get address of display reg i
MOV     @(SP)+,-(SP)    ; get value of display reg
ADD2    x,(SP)          ; add in variable offset x
```

Full stack machine support would allow variables to be referenced uniformly regardless of their lexical level. Because of the limited number of registers on the VAX, the entire display cannot be contained in registers.

5.2.2 Uplevel Parameter References

By VAX conventions, invocations using the CALLS instruction have the argument pointer register set to a block of storage that contains the subprogram's actual parameters. This is shown in Figure 5-3.

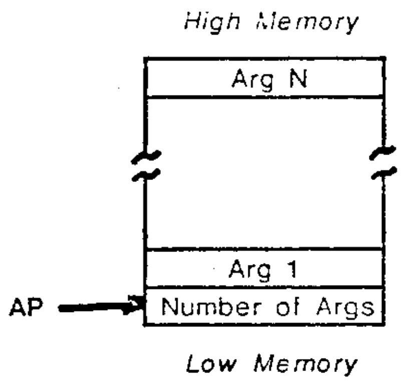


Figure 5-3: Actual Argument Block for a Subprogram

This convention is followed by nearly all VAX software[9]. Our philosophy dictates that we remain compatible with this convention.

One problem with parameters is uplevel referencing. The display can be used to calculate the

[9]The Unix Shell does *not* use this convention when it passes arguments from a command line. Special code must be generated when Ada programs are used as Shell commands.

address of variables in enclosing scopes, but not of the parameters to subprograms in enclosing scopes. Consider the following Ada program fragment:

```
procedure Outer(X: Integer) is
    procedure Inner is
        ... X ...
    end Inner;
...
end Outer;
```

Consider the stack for a call of Inner (shown in Figure 5-4).

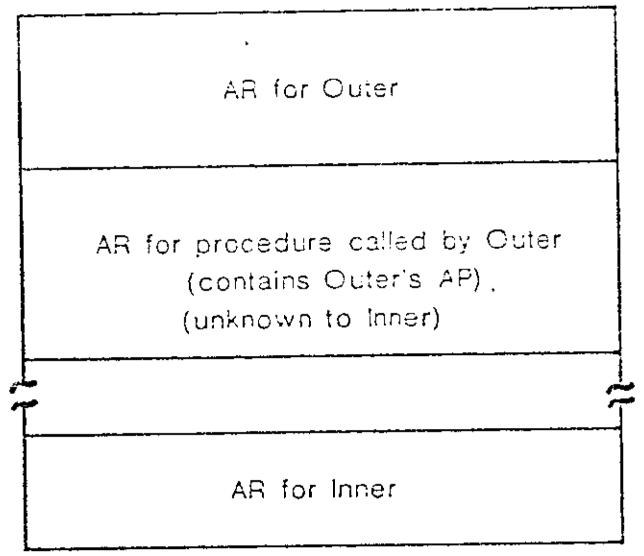


Figure 5-4: Partial Stack for Nested Subprogram Call

When the reference to X is encountered, it is unknown where the argument pointer for Outer is stored. The current argument pointer (AP) is for Inner. Although the display can give the frame pointer for Outer, the saved AP before the activation record for Outer is the argument pointer of Outer's caller. Outer's AP is saved when it calls another subprogram (which directly or indirectly calls Inner). To find the AP for Outer requires a stack analysis to determine the activation record for the subprogram called from Outer, and then to retrieve the saved AP. This is too much overhead, even if the situation occurs infrequently. A better solution is to save the *current* argument pointer in the current activation record, an overhead of one instruction per subprogram call and one longword per activation record. The correct AP is found by the ordinary uplevel reference mechanism, and the parameter is referenced by the standard parameter access mechanism.

5.2.3 Function Return Objects

The stack machine model leaves the results of operations on the stack. However, function calls use the stack for passing parameters and saving state. One cannot merely push the returned value on the stack and leave; the function return pops the stack, including the returned value. The VAX conventions circumvent this problem by keeping returned values in register 0. After the return statement is executed, the contents of register 0 are pushed on the stack by the calling subprogram. The function result appears to be left on the top of the stack correctly.

This technique fails for objects that cannot fit in a register. When a large object is passed as a parameter, only the address is passed in the call by reference scheme, so it would seem that an address could be returned as the value for a large object. When passing an address as a parameter, the storage for the Ada object has already been allocated as part of the current activation record. When a value is returned from a function, the object does not exist until *after* the function has been called, and so is part of the function's activation record. The address can be returned but the Ada object to which it refers does not exist after the function call. Nor can storage be allocated before the function call since the size of the returned object may not be known before the function returns. This is an inherent property of Ada and not of the stack model or the VAX architecture.

A trick is needed to prevent the function's activation record from disappearing when the function returns. This allows the returned object to remain on the stack. The RET instruction, which is used to return from a function call, automatically restores the stack pointer to its value before the function was called. To move the SP back to its pre-RET value, it must be saved in a location that will not be altered by the RET instruction. Register 1 serves this purpose. The code sequence for leaving a function is:

```
MOV   (SP)+,R0    ; save object's address
MOV   SP,R1       ; save the SP value
RET
```

and the code following the CALLS instruction at the caller's site is:

```
(CALLS)
MOV   R1,SP    ; restore the SP to save object
PUSH  R0       ; place object address on top of stack
```

There is another aspect to the returned object problem: evaluation order is changed. Consider the assignment statement:

```
M := P;
```

where M and P are arrays. The normal VAX code generated for the stack machine is:

```
PUSHA M
PUSHA P
MOVC3 <size of array>,(SP)+,(SP)+
```

that is, push the addresses of M and P, then do a block transfer (MOVe Characters with 3 arguments), popping the addresses. This paradigm breaks down in the case of returned objects. Consider the statement:

```
M := F(P)
```

where M and P are arrays, and F is a function that returns an array of unspecified size. The simple VAX code sequence is[10]:

```
PUSHA   M      ; push address of array M (part of assignment stmt)
PUSHA   P      ; push address of array P (param to F)
CALLS   #1,F   ; call function F with 1 arg
MOV     R1,SP  ; restore the stack pointer for returned object
PUSH    R0     ; place reference to returned value on top of stack
MOVC3   <size of array>,(SP)+,(SP)+
```

The block move instruction at the end of the code sequence references the top two longwords on the stack as the addresses for the block move. But the destination address is deep in the stack, below the returned object. The second operand to the block move instruction is garbage. In these cases, the Ada statement is rewritten (by XFORM) into the following pseudo-Ada statement:

```
F(M) =: A
```

which means reverse assignment. The semantics are: push the left hand expression (the returned object), push the right hand expression (the target for the assignment), and assign the left hand side to the right hand side. Temporary variables and an extra statement to save the left hand side ensure that the assignment statement is elaborated correctly.

The returned object problem is pernicious. It affects many other parts of the abstract machine and run-time system. It is ill supported by the VAX, and we speculate, by most other machines. The obvious solution is exclusive use of the heap for all such dynamic objects. Heaps are not well supported by most machines, which limits the effectiveness of the solution.

[10] To simplify the example, all constraint checking is omitted.

5.3 Implementing Exceptions

The VAX architecture provides a uniform exception handling mechanism, although it is not completely consistent with the requirements in the Ada rationale. The rationale states:

> One important design consideration for the exception handling facility is that exceptions should add to execution time only if they are raised. [16]

In the rationale's example implementation, no execution-time overhead is needed to enable or disable exception handlers.

The VAX architecture specifies that the address of the current exception handler is in the current activation record, and since this data structure does not exist until run time, there must be some execution overhead for the enabling of an exception handler.

The implemented system uses a small amount of run-time overhead. This is considered acceptable as it effectively utilises the VAX hardware and architecture rather than producing convoluted code to circumvent it. There are four parts to the exception handling mechanism:

- Translating the exceptions
- Translating the exception handlers
- Enabling and disabling the exception handlers
- Raising exceptions and propagating raised exceptions

5.3.1 Translating Exceptions

Although exception names follow the same scope rules as other names, the exceptions can be propagated beyond the scopes in which they can be named. This requires exceptions to have unique program-wide identification, even with separate compilation. This is accomplished by assigning each exception a static location in memory. Because static variables have unique addresses, these addresses may be used as unique identifiers.

5.3.2 Mapping Exception Handlers

Exception handlers are blocks of code. When an exception is raised, the run-time system will execute a JMP instruction to the correct handler. Because all the environment registers are set to the correct values before this transfer, the exception handler appears to be executing in the scope where it is defined. Handlers finish their execution by either a JMP, if they are attached to a block, or by a RET instruction if they are attached to a routine.

The only subtle issue is changing the current exception handler. When an exception handler starts execution, it first changes the pointer to the current exception handler in the activation record. If this were not done, an exception raised in the currently executing exception handler would be routed back to the exception handler that raised it, causing a infinite loop (in addition to being an incorrect implementation of Ada).

5.3.3 Enabling Exceptions

The VAX architecture states that the location pointed at by the frame pointer (FP) should contain the address of the current exception handler. Whenever a scope that has an exception handler is entered, the code generator will generate an instruction that loads the address of the current exception handler. When the scope is left, the address of the enclosing exception handler (up to the routine level) will be placed into the specified location. These addresses are known at compile time, so the maximum overhead for having an exception handler that is never executed is two instructions per block or routine. By convention, an address of 0 is taken to mean that no exception handler is enabled for the current routine. Since the CALLS instruction automatically sets the predetermined location in the activation record to 0 on routine entry, there is *no* additional overhead for routines that have no exception handlers.

5.3.4 Raising Exceptions

There are two ways that the implementation recognises exceptional conditions: hardware recognised traps and software calculated values.

5.3.4.1 Hardware Exceptions

Hardware exceptions are communicated by the Unix *signal* mechanism. When a program starts execution, our run-time system enables a routine to catch all possible signals. When a hardware exception occurs, the Unix system maps it into a special code and calls the designated routine. This routine translates the Unix signal code into an Ada exception value, saves this value in the task control block (where it may be interrogated by an exception handler) and then calls the run-time routine for handling Ada exceptions. This run-time routine unwinds the stack, looking for an enabled exception handler. The RET instruction is used to perform the unwinding which guarantees that the state is completely and correctly restored. By enclosing the entire program in a block that contains a default exception handler, any uncaught exceptions will be processed by the system.

This method works well, except that Unix does not relate all the possible exception information that the VAX generates. For example, the underflow, overflow, subscript range, and divide-by-zero exceptions are mapped into a single arithmetic signal. This is too imprecise a reporting mechanism. A modified signal routine is installed in our Unix kernel to report the exact hardware exception that the VAX finds.

We used the INDEX instruction both to perform subscript calculations and to check range constraints on assignment to discrete types. It caused us some difficulty to distinguish these two cases, since both generate the same fault. It would have been possible to detect the difference between a subscript calculation and a range check by examining the INDEX instruction that generated it. However, after a subscript fault the program counter points to the instruction *after* the INDEX that caused the fault. The VAX has variable-length instructions, and lacks any form of "instruction length" indicator that could be examined after a fault. This makes it very difficult to try to "back up" over the instruction. We finally decided to follow each subscript-checking INDEX with a no-operation instruction. The run-time routine that handles machine-check faults examines the instruction after the fault in order to determine whether to raise `RANGE_ERROR` or `INDEX_ERROR`.

5.3.4.2 Software Exceptions

To make a uniform mechanism for handling exceptions, all software exceptions are made to look like hardware exceptions by execution of a Unix signal. Because all software generated exceptions are mapped into the same code by Unix, it is necessary to have some convention to indicate which exception has been raised. This is done by storing the exception in the task control block *before* making the signal call. When a software signal is raised, the signal handling routine assumes that the correct exception value is already present in the task control block.

5.3.5 Propagating Exceptions

Reraising an exception is just a Unix signal call, like any other software exception. It is easier than other software exceptions, because the current exception is already stored in the task control block.

6. Working Notes

This chapter contains modified versions of some of the working notes created during the Charrette design stages. They typically describe things not implemented by the end of the project. Because they are working notes, rather than finished documents, they may not be completely accurate, and may ignore important or subtle points. We present them solely in hope that they might save some duplication of effort on the part of other implementors.

6.1 Generics

6.1.1 Expansion of Generics

Our original plans called for another phase before TYPEREP, to be called GENERIC. GENERIC expands Ada generic instantiations by a process similar to macro expansion. We chose this over more ambitious schemes because of its simplicity. The remaining phases can ignore generics completely, thus simplifying their structure.

The generic expansion scheme is not really as simple as macro expansion. When the body of a generic program unit refers to an **in** formal parameter, for example, the instantiated body must use the value of the actual parameter at the point at which the unit was instantiated. Thus for each **in** parameter, GENERIC introduces a **constant** variable declaration where the variable is initialised to the actual parameter expression.

The distinction in $TCOL_{Ada}$ between built-in operators and user-defined subprograms causes another problem. In $TCOL_{Ada}$, a call on a built-in function is represented by a special operator, such as PLUS_OP for the built-in addition operators. Calls on user-defined subprograms are all indicated via the single CALL operator; additional fields within the tree node point to the symbol table entry for the user-defined function. Calls on a **function** parameter to a generic are represented in $TCOL_{Ada}$ via nodes containing a CALL operator. If an instantiation passes a built-in function such as integer addition, the CALL operators to this parameter must be replaced by integer addition operators.

6.1.2 Other Schemes

There are a number of more clever schemes for handling generics; we briefly considered some of them during the early design stages of the Charrette. All of these schemes are intended to reduce the number of code bodies produced by multiple instantiations of a common generic unit.

The basic notion is to generate a single code body for a generic procedure or package, and to pass some representation of the generic parameters as implicit parameters to the instantiated procedure, or to each procedure in the instantiated package. Some generic parameters are easy to deal with in this way. Expressions or objects, for example, are trivial. Procedures are harder, in that they require some representation of the environment in which they are declared in addition to the address of the procedure code body. However, there are well-known techniques for doing this [1, 24].

A more difficult part is passing types as parameters. A type parameter to a generic behaves as a private type or limited private type within the generic body[11]. This means that the most the generic body can do is declare, pass as parameters, assign, and compare for equality. Our type and subtype descriptors provide all the information needed for these operations. Thus, a descriptor can be passed to represent the type parameter.

Analysis of the body of a generic package may reveal that some procedures do not require all of the generic parameters. In this case, the parameters could be eliminated. Alternately, the entire set of generic parameters could be held in a record structure, and a pointer to this structure could be passed as a parameter.

6.2 Block/Procedure Entry/Exit Code

This section gives some explanation of the code sequences generated for scope entry and exit. The sequences shown represent the worst case (maximal amount of code generated). There are many cases where CODE will avoid generating unnecessary code. There are other cases where a certain amount of analysis in the code generator would allow substantial savings of code but the effort to include such tests would not serve our purposes. Recall that SP is the stack pointer, FP is the frame (current activation record) pointer, TP is the (currently active) task pointer, AP is the argument pointer, R0 is general register 0, and R1 is general register 1.

6.2.1 Block Entry Code

This is the sequence for entering a block.

```
MOVL    SP,a(FP)        ;Save the stack pointer for leaving the block
                        ;Clear out the local storage with zeros
MOVC5   0,(SP),0,<block storage length>,<block storage offset>
<elaborate declarations>
MOVAL   Addr1,(FP)      ; enable exception handler
```

[11]This statement is true only in preliminary Ada. In revised Ada, the generic definiton may indicate the class of permissible types for a particular parameter. In this case, the generic body may make attribute inquiries on the type.

6.2.2 Block Exit Code

Code for leaving a block, GOTO out of a block, or EXIT from a loop.

```
<push returned value, if any, onto stack>
MOVL    (SP)+,R0        ;Save returned value (if any) in Reg 0
MOVL    b(FP),SP        ;Restore stack to before block allocations
PUSHL   R0              ;Restore the returned value from Reg 0
MOVAL   Addr2,(FP)      ;Restore old exception handler
JBR[12] Addr3           ;Transfer if not falling out of block
```

6.2.3 Procedure Call Code

Subprogram call.

```
<push arguments in reverse order>
CALLS <#args>,<routine> ;The actual call instruction
MOVL  R1,SP             ;Restore stack pointer to find returned object
PUSHL R0                ;Restore returned value to top of stack
```

6.2.4 Procedure Exit Code

A **return** statement.

```
<push returned value onto stack>
MOVL    (SP)+,R0        ;Save returned value (if any) in Reg 0
MOVL    c(FP),d(TP)     ;Restore old display value
MOVL    SP,R1           ;Save current stack pointer for returned object
RET                     ;Pop stack and return to caller
```

6.2.5 Procedure Entry Code

Subprogram definition. AR means Activation Record.

```
MOVL    e(TP),f(FP)     ;Save old display pointer in current AR
MOVL    FP,e(TP)        ;Save current AR pointer in display
MOVL    AP,g(FP)        ;Save current argument pointer in AR
                        ; for uplevel addressing of parameters
                        ;Allocate stack storage.  Stacks grow downward
SUBL2   <#bytes required for all block>,SP
MOVC5   0,(SP),0,<block storage length>,<block storage offset>
                        ;Clear out the local storage with zeros
<elaborate declarations>
MOVAL   Addr4,(FP)      ;Set up exception handler
```

[12] This is a UNIX assembler extended mnemonic. If the destination of the jump is within the range of a short-form branch, the short form will be used. Otherwise a long-form JMP will be used.

6.3 Motivation for the Stack Frame Layout

This section refers to the stack frame layout shown in Figure 5-1.

Everything from the word pointed to by FP up to the beginning of the AR is done by the CALLS instruction.

A display is used for uplevel references. The current level references could be made through the frame pointer if we were willing to have the CODE phase recognise this special case. The overhead of display updates seems smaller than one uplevel reference via static links. Because no procedures are being passed as parameters or variables, a single word of saved display is sufficient. To give a firm number, I suggest we use 64 display registers.

Blocks internal to a subprogram are squashed to be on the same lexical level (in implementation only). This saves overhead of the AR from the (FP) to the beginning, updating display registers, and uplevel addressing of the subprogram's global variables within the block. Care must be taken to not leave garbage in the fixed area between block entries as a nonpointer value in one block may be a pointer storage location in the next. If not cleared, when the garbage collector is called in the inner block, the pointer storage location could be interpreted as an address and destroy the program.

Because the VAX addressing modes only allow autoincrement/autodecrement on registers, the SP is in a register. The old stack pointer values represent the top of stack before a block was entered. To pop the stack of a block's variable-sized storage, one resets the SP to the top of stack before the block was entered.

Some obvious optimisations come to mind:

- If there are no access variables in a procedure, leave the Garbage Collection Temporary Pointer zero and never update it.
- If no exception handler is declared in a block, do not update the exception handler entry.
- If a procedure does not call other subprograms, do not bother changing the display.
- If there is no variable part for a certain block level, remove its old stack pointer storage and do not alter the SP on entry/exit from that level.

6.4 Default Initialisation

Our system currently handles default initialisation of access variables (i.e., pointers) to null by clearing memory to zero when it is allocated, and using zero to represent null.

There are two ways to handle default initialisation of records under our scheme:

- Augment the record type descriptor with initialisation information, and have the allocator perform initialisation from this information. For each field of the record, we would have a pointer to an auxiliary variable holding the initial value for that field. If the field has no initial value, the pointer would be null. For a field that is a composite object, i.e., an array or a record, the allocator would follow the pointer to the field's type template to call itself recursively.
- Build a procedure, at the same lexical level as the type declaration, that initialises the fields. For a composite object, the procedure would call the initialisation procedure(s) for the component type(s). Such as scheme is described by Holt and Wortman for the Euclid language [13].

In either case the actual initialisation expressions must be evaluated at the point of the type definition, and stored in auxiliary variables.

The two approaches are functionally equivalent. If we have the compiler build the subroutine, we should probably keep in mind what the structures would be like if the allocator were to do it. XFORM might be able to generate code that mimics the way the allocator would walk through the descriptor structure.

It is tempting to consider another potentially more efficient method for handling default initialisation of records. Just after the record type declaration is elaborated, allocate a single auxiliary variable of the record type and fill it in with the initial field values. When a variable of that record type is allocated, we would initialise it by block copying the auxiliary record variable into it. However, there are complications for records with discriminants or task components. For records with dynamic arrays, different instances can have different sized arrays. For variant records, fields in the variant component lists may have initial values, thus the initial value of a record variable (and, of course, possibly the size) will depend on how the variable is constrained. Record discriminants may also be used as discriminant constraints on an inner record field (see Section 2.1.5). In revised Ada, tasks are activated implicitly, either just after allocation for tasks on the heap or just after the "**BEGIN**" for a tasks local to a block, subprogram, or package (see [17], section 9.3). Thus, the initialisation of a task or of a record or array with task components must arrange for the activation of the task(s).

6.5 Heaps and Secondary Stacks

There is a problem lurking in the the VAX/Unix environment which we managed to avoid only because tasking is not implemented. Because interprocess communication using Unix pipes is far too slow to allow intertask communication in Ada, it is necessary to execute all tasks in the same address space. Such an implementation requires one stack for each thread of control. One would like the hardware to check for stack overflow in the course of normal instruction execution. This is not difficult to do on the VAX. If the page registers are set to prevent any access at the stack boundary, when the stack grows into that page, the hardware will catch it. Unix does not allow a process to specify which parts of memory are to be used for what purposes. The process is restricted to data, text (i.e., read only), and stack segments. Additional "stacks" may be grown from either the data or stack segments. Every time data are pushed on the stack, the stack pointer must be checked to ensure that one stack has not overlapped another. This is unacceptable overhead in a stack machine. We do not have a good solution to this problem.

This problem should *not* impact the implementation of heaps for dynamically allocated objects. Because heap objects are allocated infrequently and at well defined places, the checks to ensure that the heap has not overrun the stack are easy. By keeping the heap in data space and the stack in stack space, Unix will guarantee that the two areas do not overlap.

Naturally, one could avoid the problems of multiple stacks by allocating each activation record for each procedure from the heap. The size of the fixed part is known at compile time, and the amount of extra space needed for expressions can be found by stack-height simulation. The dynamic portion of the activation record would also be allocated from the heap. The problem of cactus stacks for tasking vanishes under this scheme, and "stack overflow" becomes running out of heap storage. We regard this as excessively expensive and prefer to use the page registers in the VAX.

6.6 Optimisation

6.6.1 TCOL_{Ada} Optimisation

There is one form of the PQCC DELAY phase that performs TCOL-to-TCOL optimisation transformations. It should be relatively easy to incorporate into the Charrette; we would have to add a few more fields to our TREE_NODEs, to hold some extra information produced by this phase.

6.6.2 MIL Optimisation

We should be able to buy a fair bit by adding a MIL optimiser phase following XFORM. The type and subtype descriptors are all constant, in the sense that once they are initialised they are not modified. In MIL they look just like ordinary variables. We can add a CONSTANT attribute to VAR nodes, and have XFORM set this attribute to TRUE for descriptors and FALSE for other VARs. If the initial value for a field in the descriptor is a compile-time constant, then the optimiser can replace references to the field by the actual constant. This would, in particular, allow PEEP to optimise away range checking on integers, since it would be able to see that the bounds on the INDEX instruction were the maximum and minimum integers. Given the number of range checks, this particular optimisation would be very useful.

7. Conclusion

7.1 History

The original versions of the parser and semantic analyzer were developed in mid 1978. Beginning in June 1979 these were extensively modified for the Ada language. At this time approximately 1/3 of the code and most of the design philosophy was retained. In four months, with an average of three people working full time (2000 person hours), a working version was released. Since that time more and more language features have been implemented. In August 1980 changes for revised Ada were started. Because modularity and flexibility were always part of the design, we needed no major revisions when the final report on revised Ada was published. As with any large evolving program, there are areas that could stand being rewritten. However, the emphasis on modularity has resulted in good readability and maintainability. Improvements continue to be made in the user interface, error recovery, capacity, and performance.

Work on the Back End began towards the end of October, 1979. Most of November was spent on design of MIL, the run time-system, and the phase breakdown of the compiler. The project was suspended in December because of end-of semester academic duties[13]. By the second week of January, the Back End was producing code for trivial programs. Work continued until early May, 1980, when the project was terminated. The current system thus took four people six months, working about half time, for a total of twelve person-months.

7.2 Performance

The compiler front end consists of about 260 Tops-20 file pages[14] of source code. Both programs run on DEC Tops-20 and Tops-10 systems. The parser can be run separately or as part of the entire front end as a Tops-20 dependent fork. A sample 400 line program, which included approximately 20 procedures, required 150 seconds of CPU time on a Dec-System 20. A large amount, almost 1/3, of this time was spent in garbage collection. Semantic analysis alone, without $\mathrm{TCOL}_{\mathrm{Ada}}$ production, speeds this up slightly because of extensive character string processing required for generating $\mathrm{TCOL}_{\mathrm{Ada}}$. The executable version of the parser takes 50 Tops-20 memory pages and the semantic analyzer / $\mathrm{TCOL}_{\mathrm{Ada}}$ generator takes 210 pages. This does not include the symbol tables and symbols for the predefined STANDARD package which adds another 40 pages, nor those for TEXT_IO. The current symbol table capacity is approximately 1500 symbols beyond those in STANDARD and uses a

[13] Several of the authors had teaching responsibilities.

[14] A Tops-20 file (and memory) page contains 512 36-bit words.

total of 400 pages. These statistics reflect the September 1980 version and are of course subject to change.

It is difficult to accurately measure the speed of the Charrette Back End. The present implementation writes an ASCII version of the internal state into a text file between each phase. This imposes a huge overhead on each phase that is an artifact of the support package we happened to be using. In a production version of the compiler, the data would have remained in core.

The speed of the compiler is shown in Figure 7-1. The input in this case was a 12-page, 239-line desk calculator program. All times are in seconds. Times are shown as CPU time and corresponding real time on a lightly-loaded system. The "run time" columns shows the performance of the phases after elimination of I/O time. Times are given for both the debugging version of the compiler (the one we used during development) and for a special version with the debugging overhead factored out. No CPU time is shown for the OUTPUT phase, or for output time for other phases, since it was less than the resolution of the timer (about 10 milliseconds).

	Run Time			Input Time		Output
Phase	Debug	NoDebug	Real	CPU	Real	Time
parser	.01		9			
semantic analyser	64.00		200			
TypeRep	.43	.11	4	32	157	35
Xform	1.81	.45	12	35	161	39
Block/ActRep	.98	.25	6	21	92	51
Code	8.78	2.26	41	28	159	118
Output	-----	---	---	90	721	83

Figure 7-1: Compiler Speed

These figures indicate that the Back End would process about 4600 lines per CPU minute for a "production" version. Assuming a production front end of roughly equivalent speed would result in a 2300 lines-per-minute compiler.

The size of the Back End is given in Figure 7-2. Code and Data sizes are given in terms of 36-bit PDP-10 words; source code sizes are shown in lines. The first three entries show the size of the support routines which are present in all phases. The remaining entries show the sizes of phase-specific routines; the last column for each such entry shows the total size of the phase, including the support routines. Other experience indicates that turning off debugging in the compiler cuts the size of the code in half, partly by removing debugging checks and partly by allowing the BLISS compiler optimiser to work [33]. Thus, a production version of the back end would be about 50K words in size.

Support Routines	Size	
	Code	Data
LG support	12879	3234
Debugger	12984	2840
Other support	2613	334
	-----	----
	28476	6408

Phases	(without support)		(with support)		
	Code	Data	Code	Data	Source
Typerep	9838	2457	38314	8865	2123
Xform	23599	1315	52075	7723	7189
Block/ActRep	5968	2040	34444	8448	2062
Code	23838	4090	52314	10498	4033
Output	7691	527	36167	6935	1100
	-----	----			-----
	99410	16837			16507

Figure 7-2: Back End Size

7.3 Retrospective

7.3.1 Intermediate Languages

7.3.1.1 TCOL

We found TCOL$_{Ada}$ to be reasonably easy to use, despite the fact that we had little influence on its design. We feel it is important for an intermediate language at this level to accurately reflect the original source program. Performing canonicalisations is reasonable and desireable, but it should not go so far as to destroy information.

There were some rough spots in the TCOL$_{Ada}$ we got as input to the back end. In some cases these were TCOL problems, in some cases they were canonicalisations that TCOL permitted the Front End to do. An example of a TCOL problem was that TCOL$_{Ada}$ distingushes between built-in operators and user-defined subprograms. This caused difficulties in places where we would have liked to treat the two uniformly. An example of a canonicalisation problem was that at one time the Front End expanded multidimensional arrays as arrays-of-arrays. Because of dope vector information, these two are not equivalent in run-time cost, although they are functionally equivalent.

An example of a useful canonicalisation is having the Front End expand a type declaration into a type declaration and a subtype declaration; this transformation is mentioned in Section 4.2.1.

7.3.1.2 MIL

We believe that the decision to use MIL was correct for the needs of our project. It allowed us a clean separation of work in the compiler, and made it possible to construct the two largest phases (XFORM and CODE) in parallel. It also allowed us the possibility of retargeting the back-end to produce code for another machine.

For a longer-term project, it would probably be better to retain and decorate the TCOL_{Ada} tree, rather than translate and replace it as we did. MIL is too low-level to allow the code generator to emit high-quality code. As we developed the compiler, we found that we had to keep adding more high-level information to MIL to describe range constraints, exception handling, and function return results. We had to embed in the CODE phase knowledge of the array descriptor layouts, in order to allow it to generate reasonable subscripting code. We could have further improved the code by embedding knowlege about record field accessing in MIL, such as by adding a field-access operator. At the moment, field access is represented by combinations of addition and indirection; this obscures what is going on from the code generator, and prevents it from finding better code sequences.

The one exception to this is for representing data structures. We believe that MIL's data layout description is better for code generation than the corresponding TCOL_{Ada} structures; this is primarily because the TCOL_{Ada} structures were designed to be close to Ada, whereas the MIL structures are closer to machine level without being completely machine-dependent.

7.3.2 Implementation Languages

The classes and virtual procedures of Simula provided a good basis for building the Front End semantic analyser. They allowed a great deal of flexibility, and helped in hiding non-essential information from the designer. Their disadvantage is that the resulting program is large and slow.

We chose BLISS-10 for the Back End primarily because of the existence of a package for manipulating the Linear Graph Notation in which TCOL_{Ada} is expressed. The LG package was of great benefit to us. The ability of the reader/writer to handle all input and output automatically depended on violating type safety, in a manner analogous to storage allocators; such a package might be difficult to construct in a typed language like Ada. On the other hand, BLISS' lack of a type system hurt us. We did not make the characteristic BLISS errors (missing a dot, adding an extra semicolon, violating typing, passing the wrong number of parameters to a routine) very often, but when they happened they required a great deal of time to find.

The slowness of the BLISS compiler was often a problem. We would often lose an afternoon from one of the phases because any bug fix would require several hours of clock time in recompilation and relinking. We avoided using BLISS-36, a technically better language than BLISS-10, because at the time we began the project the BLISS-36 compiler was bug-ridden and several times slower than BLISS-10.

7.3.3 Tools

We feel that it is very important for even a medium-scale project like ours to have a good set of tools. The LG package was very useful, but the reader/writer was very slow. The ASCII form of the intermediate languages allowed us to debug individual phases before their predecessors were working, but the length of time it took to read in the input to a phase meant that debugging was slow and painful.

Towards the end of the project we developed a MIL pretty-printer that made the compiler output much more readable; we would have saved a lot of debugging time if it had been written earlier. We developed a peephole optimiser to reduce the size of the final assembly language output, primarily to help debugging.

7.3.4 Ada

We managed to confirm that many implementations suggested in the Rationale were reasonable. There are, however, a number of places where things are more difficult than the Rationale leads one to believe.

- It took us quite a while to design run-time representations that could handle full type composition. We did manage to do so, without having to add restrictions to the facility. Our design is described in Chapter 2.

- The ability of a function to return objects whose size is not known at the call site gave us a great deal of difficulty. This particular problem permeates the whole compiler. We discuss this problem in Sections 4.4.1 and 5.2.3.

- Block comparison and copying of composite objects can essentially be done as suggested in the Rationale, but "unallocated" fields of a composite structure cause problems. We discuss this in Sections 2.3.1, 2.4.3, and 4.3.1.

- Exception handling was quite easy, but the VAX architecture helped out. We describe exception handling in Section 5.3.

- Default initialisation of records that have discriminants or that contain task components essentially requires building a procedure (or equivalent data structure to be interpreted by the storage allocator). This problem is discussed briefly in Section 6.4.

Availability

For further information about the Intermetrics Front End, contact

Mike Ryer
Intermetrics, Incorporated
733 Concord Avenue
Cambridge, MA 02138

The Ada Charrette compiler was developed at Carnegie-Mellon University to investigate the issues involved in implementing Ada; it was not designed to be used by a general user community. We do not wish to assume the burden of maintaining it, and in many ways, we believe it would be unsuitable for most prospective users. Hence, we do not intend to distribute it.

We are, however, willing to make special arrangements with other implementors of Ada who are interested in working with us and with whom there can be a useful collaboration. Even in such cases, however, there are a number of conditions we must impose to protect ourselves:

- In order to use the compiler, it is essential to first obtain a Front End that emits TCOL_{Ada}, such as the one produced by Intermetrics. CMU cannot distribute its copy of this program.
- The party with whom we make such agreements must agree to not distribute the Charrette to other organisations or to use it for purposes other than those specifically agreed upon. In effect, the Charrette is to remain the property of CMU and we will retain control of its distribution and use.
- We request specific acknowledgement in any papers or products resulting from the use of the Charrette -- either its direct use, or the use of ideas gleaned from it.

Acknowledgments

The Front End project involved many people over a two-year period. We especially wish to thank Mark Davis and David Levine for the initial design of the Red Translator from which the current semantic analyzer was derived. All of the people who worked on the program added immeasurably to its success: Peter Belmont, Ben Brosgol, Bob Cassels, John Nestor, Mike Ryer, Michael Tighe, David Spector, Mary Van Deusen, and Bruce Wilcox.

The CMU effort in Ada has been extensive and we have benefited from many peoples' time and effort. Mike Acceta hacked on Unix to allow us to implement Ada exceptions correctly. Raj Reddy provided VAX time and accounts for debugging the compiler system. Bill Wulf provided support and encouragement for the entire project from its inception. We also thank John Zsarnay, Mahadev Satyanarayanan, Steve Shafer, and Jim Gosling for the time they spent with us in tracing down bugs in VAX microcode, Unix ambiguities, and C hacks.

Richard Snodgrass provided many helpful comments on earlier versions of this document.

I. Class Hierarchy in the Semantic Analyzer

The following describes the structure of the classes in the semantic analyzer for Ada written in Simula. The main Simula classes are[15]:

EMIT_BASE
A. NODE
    1. Identifier
    2. Exp
        a. texp (type expression)
            (1) type_denote_texp
            (2) derived_def
            (3) int_def
            (4) real_def
            (5) enum_def
            (6) array_def
            (7) record_def
            (8) access_def
            (9) private_def
        b. range_exp
        c. lit_exp
            (1) int_lit
            (2) float_lit
            (3) fixed_lit
            (4) null_lit
            (5) string_lit
            (6) char_lit
            (7) enum_lit
        d. accuracy_cn
        e. t_in_not_in
        f. r_in_not_in
        g. others_exp
        h. compon_assoc
        i. name_exp
            (1) invoke_exp
            (2) array_elt_exp
            (3) dot_select_exp
            (4) name_all
            (5) attr_select_exp
        j. allocator_exp
        k. convert_exp
        l. qualified_exp
        m. paren_gel
        n. record_agg_choice
        o. when_exp
    3. DCL_NODE
        a. Obj_dcl
            (1)formal_parm_dcl
        b. Pragma_dcl
        c. Exception_dcl
        d. Use_dcl
        e. Unit_dcl
            (1) subprogram_dcl
            (2) package_dcl
        f. Entry_dcl
        g. Type_dcl
        h. Subtype_dcl
        i. Rep_specl
        j. Renaming_dcl
    4. BODY_NODE
    5. COMPIL_UNIT
    6. PRAGMA_NODE
    7. STM_NODE
        a. null_stm
        b. goto_labelled_stm
        c. exit_labelled_stm
        d. assign_stm
        e. return_stm
        f. goto_stm
        g. exit_stm
        h. raise_stm
        i. call_stm
        j. delay_stm
        k. abort_stm
        l. code_stm
        m. cmpd_stm
            (1) if_stm
            (2) block_stm
            (3) case_stm
            (4) loop_stm
            (5) accept_stm
            (6) select_stm
    8. case_node
    9. handler_node
    10. list_node
    11. stack_element
    12. stack_head
    13. integer_node
B. SYMBLOCK (Symbol Table)
    1. Formals_symblock
        a. Record_symblock
    2. Mod_symblock
C. SYMTHING (Symbol)
    1. Vrbl_sym
        a. recfield_sym
        b. formal_sym
    2. Unit_sym
        a. Subprogram_sym
            (1) entry_sym
        b. Module_sym
    3. Pragma_sym
    4. Enumeral_sym
    5. Label_sym
    6. Typemaster
        a. private_tm
        b. scalar_tm
            (1) int_tm
            (2) fixed_tm
            (3) float_tm
            (4) enum_tm
        c. array_tm
        d. record_tm
        e. access_tm
        f. task_tm
        g. pseudo_typemaster
            (1) ambig_tm
            (2) unspec_tm
    8. package_body_psym
    9. exception_sym
    10. hidden_sym
D. PRAGMA_INFO
E. TICK_INFO

[15] Paren_gel means parenthesised general expression; it is transformed into convert_exp, aggregate, invoke_exp, or index_constraint

II. Complete Example of Simula Classes for a Simple Language

A class skeleton (a class definition with only the parameters and visible (not hidden and not protected) components listed) is given for each grammar rule. The semantics given in the SEMANTIC_CHECK routines are not for Ada, but for a hypothetical Pascal-like language. **Boldface** is used to represent Simula keywords, *italics* to represent an English description of some omitted Simula code. and regular type face to indicate Simula code. Liberties are also taken when translating grammar symbols into Simula identifiers.

```
<exp> ::= <name exp>
<exp> ::= <plus exp>
<exp> ::= number
<name exp> ::= <name exp> . identifier
<name exp> ::= identifier
<plus exp> ::= <exp> + <exp>

Start Symbol: <exp>
Nonterminals: <exp>, <name exp>, <plus exp>
Terminals: number, identifier, ., +
Punctuation: ., +
```

Start Symbol: <exp>

```
class exp;
virtual: boolean procedure semantic_check;
begin
    integer this_exp_type;
end;
```

<exp> ::= <name exp>

```
exp class name_exp;
begin
end;
```

<name exp> ::= <name exp> . **identifier**

```
    name_exp class dot_exp(left_part,identifier);
    ref(name_exp) left_part; text identifier;
    begin
        boolean procedure
                    semantic_check(type_context);
        integer type_context;
        begin
            if left_part.
                    semantic_check(type_context)
                then begin
                    find identifier in symbol table
                        for left_part;
                    this_exp_type :=
                        found identifier symbol.
                        symbol_type;
                    semantic_check :=
                        (type_context =
                         this_exp_type);
                end
                else semantic_check := false;
        end;
    end;
```

<name exp> ::= identifier

```
    name_exp class identifier_exp(identifier);
    text identifier;
    begin
        boolean procedure
                    semantic_check(type_context);
        integer type_context;
        begin
            find identifier in current symbol table;
            this_exp_type :=
                  found identifier symbol.symbol_type;
            semantic_check :=
                 (type_context =
                  this_exp_type);
        end;
    end;
```

<exp> ::= <plus exp>

```
    exp class plus_exp;
    begin
    end;
```

<plus exp> ::= <exp> + <exp>

```
plus_exp class add_exp(left_exp,right_exp);
ref(exp) left_exp, right_exp;
begin
    boolean procedure
                semantic_check(type_context);
    integer type_context;
    begin
        if left_exp.
                semantic_check(type_context)
                          and
                right_exp.
                semantic_check(type_context)
            then begin
              this_exp_type :=
                 left_exp.this_exp_type;
              semantic_check :=
                 (type_context =
                  this_exp_type)
                                 and
                 (left_exp.this_exp_type =
                 right_exp.this_exp_type);
            end
            else semantic_check := false;
    end;
end;
```

<exp> ::= **number**

```
exp class number(number_text);
text number_text;
begin
    boolean procedure
                semantic_check(type_context);
    integer type_context;
    begin
      convert text into internal numeric representation;
      this_exp_type := if text is an integer
             then the_predefined_integer_type
             else the_predefined_float_type;
      semantic_check :=
            (type_context = this_exp_type);
    end;
end;
```

III. Relationship between Simula Classes and Ada Records

This appendix is intended to provide a simple explanation of the Simula class mechanism in terms of Ada. It is not intended to be a complete description of Simula or Ada. The reader is assumed to have some knowledge of Ada. Many irrelevant details of both Simula and Ada are ignored. Full details of Simula can be found in reference texts for the language [2, 28].

A Simula class is similar to an access type to a variant record in Ada. The components of a class (including its parameters) correspond with the fields of a record. Because these records exist only after they are explicitly allocated, the classes are actually access types. For example, consider the following Simula and Ada declarations:

```
class Outer(X);
    Integer X;
begin
    Integer Y;
    Y := 2
end

type OuterDummy(X:Integer) is
    record
        Y: Integer := 2;
    end record;
type Outer is access OuterDummy;
```

In both languages, an object of type Outer must be explicitly allocated at run time and some initial value filled in for the parameter. The values in the class instance/record object may be read by the selector operation, i.e., by using the dot notation.

Subclasses in Simula provide a way to declare variant records. A class declaration of the form:

```
Outer class Inside;
begin
    Boolean Q;
end
```

has a correspondence in Ada of

```
type OuterSubclasses is
        (NoSubclass, InsideSubclass);

type OuterDummy(
    Subclass: OuterSubclasses := NoSubclass;
    X: Integer) is
    record
        Y: Integer := 2;
        case Subclass is
            when InsideSubclass =>
                Q: Boolean;
            when others =>
                null;
        end case;
    end record;

type RefToOuter is access OuterDummy;
subtype Outer is RefToOuter;
subtype Inside is
     RefToOuter(Subclass => InsideSubclass);
```

Adding a new subclass in Simula corresponds with adding another variant record field in Ada. The primary difference, for defining new fields, is that Ada requires all variants to be declared with the type while Simula allows new variants to be introduced in separate subclass declarations.

A difference between the two languages is the definition of procedures. In Simula, a procedure may be associated with a class by defining it inside of that class as a component. An Ada record has no such provision. In Ada, the same procedure may be written but must include a parameter by which a record object may be passed.

There is less, but some, correspondence between Simula virtual procedures and Ada overloaded subprograms. When a virtual procedure is declared, it is a claim that there will exist a procedure by the same name in every subclass. At run time when that procedure is called, the subclass (i.e., variant) is examined and the correct procedure is selected for execution. In Ada terms, this is a claim that a procedure would exist for each different discriminant in the record type, and that the selection of the procedure to call would be postponed until run time when the value of the discriminant would be known, for instance, by combining all of the procedures into one procedure and selecting the correct procedure in a **case** statement.

IV. Run-Time Routines

ARTO	Start up routine, written in C, that enables all signals, initialises the storage allocator, and resets the heap allocator.
WRITECHAR	C routine that writes out a single character.
WRITENUM	C routine that writes out an integer.
READCHAR	C routine that reads in a single character.
READCHAR	C routine that reads in an integer.
HANDLER	Assembly language routine that intercepts Unix signals, determines the appropriate Ada exception, reinstalls (if necessary) the signal catcher, unwinds the stack as necessary, and jumps to the current exception handler.
ALLOC	C routine that allocates storage for dynamic Ada objects.
CATCHER	Default exception handler for the outer-most block of the program. Written in C.

Figure IV-1 shows the size of the run-time system. Sizes are given in 8-bit bytes.

Routine	Code	Data
ARTO	236	0
input/output	284	8
HANDLER	116	12
ALLOC	2856	40
CATCHER	68	320

Figure IV-1: Run-time Routine Sizes

V. Sample Output

This example illustrates the block and subprogram entry and exit code for a simple program.

V.1 Ada Program

```
ADA TRANSLATOR - 1980-05-02

procedure small is
        x: integer range 1..10;
begin
        x := 2;
        declare
                y: integer range 1..2;
        begin
                y := x;
        exception
                when overflow => x := 1;
        end;
end small;


              NO PARSE ERRORS
              NO SEMANTIC ERRORS
              NO SEMANTIC WARNINGS
```

V.2 TCOL_{Ada} Program

This listing omits the TCOL for the package STANDARD, which itself requires considerable space.

```
NODES 59
ROOT TREE 1:
ROOT TREE 73:
1: TREE_NODE (SOURCE "PROLOG.RDP;0/1{26}") (OP SEQUENCE_OP) (XOP NOT_DEFINED) (SUBNODES 3: 4:
   5: 6: 7: 10: 11: 12: 13: 14:)
15: TREE_NODE (SOURCE "TEST.RDP;0/1{8}") (OP LEAF_CP) (XOP INTEGER_TYPE) (DEFN 16:)
17: TREE_NODE (SOURCE "TEST.RDP;0/1{12}") (OP LEAF_OP) (XOP INTEGER_TYPE) (DEFN 20:)
20: LITERAL_SYM (SOURCE "TEST.RDP;0/1{12}") (LIT_SUBTYPE 21:) (LIT_KIND INT_LIT) (LIT_NAME "2")
   (LIT_VALUE "2")
22: TREE_NODE (SOURCE "TEST.RDP;0/1{12}") (OP ASSIGN_OP) (XOP NOT_DEFINED) (SUBNODES 15: 17:)
23: TREE_NODE (SOURCE "TEST.RDP;0/1{16}") (OP LEAF_CP) (XOP INTEGER_TYPE) (DEFN 24:)
25: TREE_NODE (SOURCE "TEST.RDP;0/1{21}") (OP LEAF_OP) (XOP INTEGER_TYPE) (DEFN 16:)
26: TREE_NODE (SOURCE "TEST.RDP;0/1{21}") (OP ASSIGN_OP) (XOP NOT_DEFINED) (SUBNODES 23: 25:)
27: TREE_NODE (SOURCE "TEST.RDP;0/1{33}") (OP LEAF_OP) (XCP INTEGER_TYPE) (DEFN 16:)
30: TREE_NODE (SOURCE "TEST.RDP;0/1{37}") (OP LEAF_OP) (XCP INTEGER_TYPE) (DEFN 31:)
31: LITERAL_SYM (SOURCE "TEST.RDP;0/1{37}") (LIT_SUBTYPE 21:) (LIT_KIND INT_LIT) (LIT_NAME "1")
   (LIT_VALUE "1")
32: TREE_NODE (SOURCE "TEST.RDP;0/1{37}") (OP ASSIGN_OP) (XOP NOT_DEFINED) (SUBNODES 27: 30:)
33: TREE_NODE (SOURCE "TEST.RDP;0/1{28}") (OP SEQUENCE_OP) (XOP NOT_DEFINED) (SUBNODES 32:)
34: TREE_NODE (SOURCE "TEST.RDP;0/1{28}") (OP LEAF_CP) (XOP NOT_DEFINED) (DEFN 35:)
36: TREE_NODE (SOURCE "TEST.RDP;0/1{28}") (OP WHEN_CP) (XOP NOT_DEFINED) (SUBNODES 33: 34:)
```

```
37: TREE_NODE (SOURCE "TEST.RDP;0/1{12}") (OP CASE_EXCEPTION_OP) (XOP NOT_DEFINED)
    (SUBNODES 36:)
40: NAME_NODE (SOURCE "TEST.RDP;0/1{37}") (PNAME "--unique_name-5100") (NAMES 41:)
42: TREE_NODE (SOURCE "TEST.RDP;0/1{34}") (OP LEAF_OP) (XOP INTEGER_TYPE) (DEFN 43:)
43: LITERAL_SYM (SOURCE "TEST.RDP;0/1{34}") (LIT_SUBTYPE 3:) (LIT_KIND INT_LIT) (LIT_NAME "1")
    (LIT_VALUE "1")
44: TREE_NODE (SOURCE "TEST.RDP;0/1{37}") (OP LEAF_OP) (XOP INTEGER_TYPE) (DEFN 45:)
45: LITERAL_SYM (SOURCE "TEST.RDP;0/1{37}") (LIT_SUBTYPE 3:) (LIT_KIND INT_LIT) (LIT_NAME "2")
    (LIT_VALUE "2")
46: TREE_NODE (SOURCE "TEST.RDP;0/1{37}") (OP RANGE_OP) (XOP INTEGER_TYPE) (SUBNODES 42: 44:)
41: SUBTYPE_SYM (SOURCE "TEST.RDP;0/1{37}") (NAME 40:) (PARENT_TYPE 47:) (CONSTRAINTS 46:)
    (PARENT_SUBTYPE 3:)
50: TREE_NODE (SOURCE "TEST.RDP;0/1{37}") (OP SUBTYPE_DECL_OP) (XOP NOT_DEFINED) (DEFN 41:)
51: NAME_NODE (SOURCE "TEST.RDP;0/1{16}") (PNAME "Y") (NAMES 24:)
24: VARBL_SYM (SOURCE "TEST.RDP;0/1{37}") (NAME 51:) (VARBL_SUBTYPE 41:) (CONSTANCY
    NOT_CONSTANT) (SPECIES VARBL) (IS_PRIVATE FALSE)
52: TREE_NODE (SOURCE "TEST.RDP;0/1{12}") (OP VARBL_DECL_OP) (XOP NOT_DEFINED) (DEFN 24:)
53: TREE_NODE (SOURCE "TEST.RDP;0/1{12}") (OP SEQUENCE_OP) (XOP NOT_DEFINED) (SUBNODES 50: 52:
    26:)
54: TREE_NODE (SOURCE "TEST.RDP;0/1{12}") (OP BLOCK_OP) (XOP NOT_DEFINED) (SUBNODES 53: 37:)
55: NAME_NODE (SOURCE "TEST.RDP;0/1{30}") (PNAME "--unique_name-5059") (NAMES 21:)
56: TREE_NODE (SOURCE "TEST.RDP;0/1{26}") (OP LEAF_OP) (XOP INTEGER_TYPE) (DEFN 57:)
57: LITERAL_SYM (SOURCE "TEST.RDP;0/1{26}") (LIT_SUBTYPE 3:) (LIT_KIND INT_LIT) (LIT_NAME "1")
    (LIT_VALUE "1")
60: TREE_NODE (SOURCE "TEST.RDP;0/1{30}") (OP LEAF_OP) (XOP INTEGER_TYPE) (DEFN 61:)
61: LITERAL_SYM (SOURCE "TEST.RDP;0/1{30}") (LIT_SUBTYPE 3:) (LIT_KIND INT_LIT) (LIT_NAME "10")
    (LIT_VALUE "10")
62: TREE_NODE (SOURCE "TEST.RDP;0/1{30}") (OP RANGE_OP) (XOP INTEGER_TYPE) (SUBNODES 56: 60:)
21: SUBTYPE_SYM (SOURCE "TEST.RDP;0/1{30}") (NAME 55:) (PARENT_TYPE 47:) (CONSTRAINTS 62:)
    (PARENT_SUBTYPE 3:)
63: TREE_NODE (SOURCE "TEST.RDP;0/1{30}") (OP SUBTYPE_DECL_OP) (XOP NOT_DEFINED) (DEFN 21:)
64: NAME_NODE (SOURCE "TEST.RDP;0/1{8}") (PNAME "X") (NAMES 16:)
16: VARBL_SYM (SOURCE "TEST.RDP;0/1{30}") (NAME 64:) (VARBL_SUBTYPE 21:) (CONSTANCY
    NOT_CONSTANT) (SPECIES VARBL) (IS_PRIVATE FALSE)
65: TREE_NODE (SOURCE "TEST.RDP;0/1{14}") (OP VARBL_DECL_OP) (XOP NOT_DEFINED) (DEFN 16:)
66: TREE_NODE (SOURCE "TEST.RDP;0/1{14}") (OP SEQUENCE_OP) (XOP NOT_DEFINED) (SUBNODES 63: 65:
    22: 54:)
67: TREE_NODE (SOURCE "TEST.RDP;0/1{14}") (OP BLOCK_OP) (XOP NOT_DEFINED) (SUBNODES 66: 70:)
    (DEFN 71:)
72: NAME_NODE (SOURCE "TEST.RDP;0/1{14}") (PNAME "SMALL") (NAMES 71:)
71: SUBPROGRAM_SYM (SOURCE "TEST.RDP;0/1{14}") (NAME 72:) (KIND PROCEDURE_SUBPROGRAM)
    (BODY 67:) (IS_BUILT_IN FALSE) (IS_SEPARATE FALSE) (LINKAGE "ADA")
73: TREE_NODE (SOURCE "TEST.RDP;0/1{26}") (OP SUBPROGRAM_DECL_OP) (XOP NOT_DEFINED) (DEFN
    71:)
```

V.3 MIL Program

This is the output of a prettyprinter, rather than the actual LG form used in the compiler. The prettyprinter suppresses certain details present in the full LG form. The comments were added by hand.

```
31:     DECLARE
51:
51:        ada ROUTINE SMALL() RETURNS NOTHING
61:           DESC_1: DESC    -- StatScalar subtype descriptor for
                             -- integer range integer'first .. integer'last
53:              INT RANGE -2147483648 .. 2147483647 <- 1
53:              INT RANGE -2147483648 .. 2147483647 <- -2147483648
53:              INT RANGE -2147483648 .. 2147483647 <- 2147483647

76:           DESC_2: DESC    -- StatScalar subtype descriptor for
                             -- integer range 1 .. 10
53:              INT RANGE -2147483648 .. 2147483647 <- 1
53:              INT RANGE -2147483648 .. 2147483647 <- 1
53:              INT RANGE -2147483648 .. 2147483647 <- 10

110:          CAST @DESC_2.2 {@DESC_1.2, @DESC_1.3}
                    -- Checks that 1 is within -2147483648 .. 2147483647
111:          CAST @DESC_2.3 {@DESC_1.2, @DESC_1.3}
                    -- Checks that 10 is within -2147483648 .. 2147483647
112:          X: DESC
106:             INT RANGE 1 .. 10

51:        BEGIN
116:          X <- 2
117:          DECLARE
122:             DESC_3: DESC    -- StatScalar subtype descriptor
                                -- for integer range 1 .. 2
53:                    INT RANGE -2147483648 .. 2147483647 <- 1
53:                    INT RANGE -2147483648 .. 2147483647 <- 1
53:                    INT RANGE -2147483648 .. 2147483647 <- 2

134:             CAST @DESC_3.2 {@DESC_1.2, @DESC_1.3}
                    -- Checks that 1 is within -2147483648 .. 2147483647
135:             CAST @DESC_3.3 {@DESC_1.2, @DESC_1.3}
                    -- Checks that 2 is within -2147483648 .. 2147483647
136:             Y: DESC
132:                INT RANGE 1 .. 2
```

```
117:         BEGIN
142:            CAST Y <- @X {@DESC_3.2, @DESC_3.3}
                       -- Assigns the value of X to Y, after checking
                       -- that the value of X is within 1 .. 2
117:         EXCEPTION
146:            IF EXVAL = 9
146:                THEN
143:                  BEGIN
144:                     X <- 1
143:                  END
146:                ELSE
13:                   RAISE EXVAL
146:            END IF
117:         END
51:       END

31:    BEGIN
147:      SMALL()
31:    EXCEPTION
151:      catcher(EXVAL)
31:    END


15:
15:    allocator ROUTINE alloc() RETURNS NOTHING
15:    END

17:
17:    c ROUTINE StatVarSet() RETURNS NOTHING
17:    END

21:
21:    c ROUTINE length() RETURNS INT RANGE -2147483648 .. 2147483647
21:    END

23:
23:    c ROUTINE boolean() RETURNS NOTHING
23:    END

25:
25:    c dynamic ROUTINE rep() RETURNS LOCATION
25:    END

27:
27:    c ROUTINE val() RETURNS INT RANGE -2147483648 .. 2147483647
27:    END

150:
150:   c ROUTINE catcher() RETURNS NOTHING
150:   END
```

V.4 Assembly Code Program

Most of the comments were added by hand.

```
# VAX Assembly Language File
# Output from phase OUTPUT V1A(5) on  2 May 80
# Input file was: SMALL

        .text
# Code which makes the main program appear to be called by another
# Ada program rather than the Shell

main:
        .word   49152
        calls   $0,Lfakemain
        chmk    $1
Lfakemain:
        .word   49152
        calls   $0,_art0
        subl2   $231,sp
        movl    sp,r11
        movl    fp,(r11)
        movl    ap,-8(fp)


# Set up the predeclared package STANDARD

        movc5   $0,(sp),$0,$60,-64(fp)  # Clear out local storage
        movl    $1,-16(fp)              # Init type desc for integer
        movl    $-32768,-12(fp)
        movl    $32767,-8(fp)
        movl    $1,-28(fp)              # Init type desc for short_int
        movl    $-128,-24(fp)
        movl    $127,-20(fp)
        movl    $1,-40(fp)              # Init type desc for long_int
        movl    $-2147483648,-36(fp)
        movl    $2147483647,-32(fp)
        movl    $1,-52(fp)              # Init type desc for boolean
        clrl    -48(fp)
        movl    $1,-44(fp)
        movl    $1,-64(fp)              # Init type desc for character
        clrl    -60(fp)
        cvtbl   $127,-56(fp)
        moval   EXH8,(fp)               # Enable default ex. handler
        calls   $0,RT0
BLKE1:
        clrl    0(fp)
        ret
```

```
RTO:

# SMALL

        .word   49152
        movl    4(r11),-4(fp)          # Save old display value
        movl    fp,4(r11)              # Update display w/ new value
        movl    ap,-8(fp)              # Save the current AP for
                                       # uplevel refs
        subl2   $46,sp                 # Allocate and clear local storage
        clrq    -33(fp)
        movc5   $0,(sp),$0,$5,-25(fp)
        movl    $1,-32(fp)             # Init type desc for int 1..10
        movl    $1,-28(fp)
        movl    $10,-24(fp)
        pushal  -33(fp)                # Push address of X
        pushl   $2                     # Push value 2
        index   (sp),-28(fp),-24(fp),$1,$1,r1
                                       # Check range
        cvtlb   (sp)+,*(sp)+           # Do assignment


# Internal Block Starting

        clrq    -46(fp)                # Clear out local
        movc5   $0,(sp),$0,$5,-38(fp)  # storage (already allocated)
        movl    $1,-45(fp)             # Init type desc for 1..2
        movl    $1,-41(fp)
        movl    $2,-37(fp)
        moval   EXH3,(fp)              # Enable block's exc. hndlr
        pushal  -46(fp)                # Push address of Y
        cvtbl   -33(fp),-(sp)          # Push value of X
        index   (sp),-41(fp),-37(fp),$1,$1,r1  # Check range
        cvtlb   (sp)+,*(sp)+           # Do assignment
BLKE2:                                 # End of internal block
        clrl    0(fp)                  # Disable exc. hndlr


# End of procedure

        movl    -4(fp),4(r11)          # Restore old display value
        ret                            # return from main program
```

```
# Exception Handler in inner block

EXH3:
        clrl    (fp)                        # disable exc. hndlr.
        movl    r0,128(r11)                 # Get the current exception
        pushl   $1                          # See if OVERFLOW
        pushl   128(r11)
        cmpl    $9,(sp)+
        jeql    BR4
        clrl    (sp)
BR4:
        jlbc    (sp)+,ITE5

# Yes, is OVERFLOW, do assignment

        pushal  -33(fp)
        pushl   $1
        index   (sp),-28(fp),-24(fp),$1,$1,r1
        cvtlb   (sp)+,*(sp)+
BLKE6:
        jbr     ITE7
ITE5:

# Not OVERFLOW, do a re-raise of exception

        movl    128(r11),128(r11)           # Save excp. value
        calls   $0,_getpid                      # Unix signal call
        pushl   $16
        pushl   r0
        calls   $2,_kill
ITE7:
        jbr     BLKE2                       # End of exc. hndlr, ret to block


# System Exception Handler

EXH8:
        clrl    (fp)                        # Disable exc. hndlr.
        movl    r0,128(r11)                 # Get exception value
        pushl   128(r11)                    # Call external rout. w/ 1 parm.
        calls   $1,_catcher
        jbr     BLKE1
```

VI. Definition of MIL

This appendix serves two purposes. First, it documents the MIL which is acceptable to the final code generation phases. Any MIL input which meets these specifications will be translated correctly into VAX assembly language. In principle, other translator systems could produce MIL and use the Charrette phases as part of their back end. Second, this appendix documents the tree transformations done on the MIL by the last several phases of the Charrette compiler. This is included to help with the maintenance of those phases. Attributes flagged with with an asterisk (*) are intended for documentation of the Charrette compiler and do not describe legal MIL input.

VI.1 Notation

The TCOL for the MIL is presented in a BNF-like form, which is self-explanatory.

For every non-terminal of the form

```
<token List>
```

there is (implicitly) a production of the form

```
<token List> → <token> | <token> <token List> | <empty>
```

Non-terminal symbols which are suffixed with a ":" indicate that, in fact, what replaces the non-terminal is not a derivation of the right-hand side of its production, but the label of a node which is such a derivation.

VI.2 <MIL Tree>

```
<MIL Tree> → <Block>
```

Until we consider separate compilation, the only form an ADA program may take is a *procedure_declaration*, which is represented in MIL as a <Routine>. The BLOCK node, which is the root (of any MIL tree), is the surrounding environment for the user-defined routine. All MIL programs which are passed to the Charrette compiler must have a single BLOCK node as their root.

Within MIL, scopes delimit the lifetimes of variables and exception handlers. The semantics of MIL for block and procedure elaboration are as follows:

- Elaborate the declarations: evaluate each expression in the LOCALS list of the block or routine, from left to right.

- Enable the exception handler for this scope if present. A scope has an exception handler if the HANDLERS field contains expressions.
- Elaborate the body: evaluate each expression in the BODY list of the block or routine, from left to right.
- Disable the current exception handler if present. If there was another exception handler enabled when this scope was entered, reenable that handler.

The semantics of exceptions and their processing closely follows Ada. When an exception occurs, the nearest dynamically enabled exception handler will be elaborated. The first action of an exception handler is to disable itself and reenable the handler which was active when the current handler was enabled. Therefore, if an exception is raised in the handler, it will be propogated up the call stack, and not be a recursive elaboration of this handler.

Exceptions are raised by hardware conditions and by the MIL RAISE statement.

Unlike Ada, there is no automatic reraising of an uncaught exception. When a handler is invoked, it will elaborate whatever expressions are specified, and continue elaboration at the end of the scope in which that handler is declared. If it is desired to propogate an exception, the user must explictly place a RAISE statement in the handler. (Note that the value of the current exception is available as an expression. See <Opr> nodes, section VI.8.3.)

VI.3 <Routine>

```
<Routine> →

        ROUTINE
           (NAME <string>)
           (PARAMETERS <Var List>:)
           (LOCALS <Decl List>:)
           (TYPE <Type>:)
           (BODY <Stmt List>:)
           (LINKAGE C | C_Dynamic | Ada | Ada_Dynamic | Algol68 |
                    Asm | Allocator)
           (HANDLERS <Stmt List>:)
           (LEXLEVEL <number>)*
           (VAXLABEL <Label>:)*
           (RETURNSIZE <number>)*
           (LOCALSIZE <number>)*
           (BLOCKSTOTAL <number>)*
           (NESTINGBLOCKSMAX <number>)*
           (MILHANDLER <Label>:)*
           (TOTALSIZE <number>)*
           (BLOCKPARENT <Scope>:)*
           (HANDLERPARENT <Scope>:)*
```

NAME	The name of the routine (optional, but useful for readability).
PARAMETERS	Formal parameters, if any, to this routine (VI.5.1).
LOCALS	Nodes describing declarations local to this routine (VI.5).
TYPE	The return type of this routine (VI.4).
BODY	The executable portion of this routine.
LINKAGE	The linkage conventions to be used when calls are made to this routine. See the <Call> node, section VI.8.2, for details.
HANDLERS	A list of statements which constitute the exception handler for this routine. This may be empty.
LEXLEVEL*	The lexical level of this routine. Filled in by the BLOCK phase.
VAXLABEL*	The label which will be generated for the VAX assembly language output. This is a LABEL node in the OBJECT language. Filled in by the CODE phase.
RETURNSIZE*	The size of a returned value, if this is a **function**. Note that MIL requires a fixed returned size. Filled in by the ACTREP phase.
LOCALSIZE*	The size of allocated local variables for this routine. This does *not* include space required by nested blocks. Filled in by the ACTREP phase.
BLOCKSTOTAL*	The total number of blocks contained in this routine. Filled in by BLOCK.
NESTINGBLOCKSMAX*	The maximum nesting depth for blocks in this routine not including nested routines. If there are no blocks in this routine, it will be zero. Filled in by BLOCK.
MILHANDLER*	Label (a LABEL node in the OBJECT language) at the start of the generated code for the exception handler attached to this routine, if any. Filled in by CODE.
TOTALSIZE*	This is the total size of the activation record for this routine. It includes the size of the routine's local variables, storage for nested blocks, saved stack pointers for those blocks, and several words of constant overhead for linkage. Filled in by ACTREP.
BLOCKPARENT*	A pointer to the enclosing block. For routines, this field is always NIL. Filled in by the BLOCK phase.
HANDLERPARENT*	A pointer to the block or routine which contains the statically enclosing exception handler. For routines, this field is always NIL. Filled in by the BLOCK phase.

The DESC field (see VI.6) of each <Var> (see VI.5.1) in the PARAMETERS field must be of the form

```
        (TYPES <Type>:)
        (SIZES one:)          ! Where one: is a subtree denoting
                              ! the literal 1 (see VI.8.8)
```

(i.e., each parameter must be a single unit in size). Passing of objects is by value. If some other form of passing is desired, it must be constructed by statements within the body of the routine or the caller. See the <Call> node for an explanation of actual argument elaboration.

The intent of the LOCALS field is to describe the static portion of the run-time stack. All routines have a return type, which is at most 1 unit in size. In the case of complex objects, this may be used to return the address of the object.

VI.4 <Type>

```
        <Type> →

                TYPE
                   (SUBTYPE NONE | INT | LOC)
                   (RANGE <Stmt>: <Stmt>:)
                   (VAXSIZE <number>)*
```

SUBTYPE	Indicates the MIL type: INT (for integer), LOC (for location) or NONE (indicating no type).
RANGE	Currently, the only values allowed in the RANGE field are two expressions which describe the range of values a value of SUBTYPE INT may have. The expressions in the range are *not* evaluated. They are given to the code generator as additional information which may be used to determine the mapping between MIL integers and the target machine's integers. The RANGE is meaningless for SUBTYPEs LOC and NONE.
VAXSIZE*	The amount of storage required for this type on a VAX. It is filled in by ACTREP.

The code generator is free to choose any representation it desires for these types. For example, INT and LOC may be represented identically.

For convenience in this appendix, we define the following TYPE nodes:

```
int:        TYPE
                      (SUBTYPE INT)
                      (RANGE <Stmt>: <Stmt>:)

loc:        TYPE
                      (SUBTYPE LOC)

bool:       TYPE
                      (SUBTYPE INT)
                      (RANGE zero: one:)
```

(Note that *int:* is, in fact, a "template" node describing any integer type. *Zero:* and *one:* are nodes describing the literals 0 and 1, respectively (see section VI.8.8).

VI.5 ⟨Decl⟩

```
<Decl> → <Var> | <Routine> | <Stmt>
```

⟨Decl⟩ nodes denote declarations of variables or routines. Note that statements may be interspersed with declarations. This allows convenient implementation of complex Ada declaration elaboration.

VI.5.1 ⟨Var⟩

```
<Var> →

          VAR
             (NAME <string>)
             (DESC <Desc>:)
             (LIFE STATIC | LOCAL | PARAMETER)
             (DOT-NODE <Exp List>:)*
             (OFFSET <number>)*
             (LEXLEVEL <number>)*
             (SIZE <number>)*
             (VAXLABEL <Label>:)*
             (TYPEDESC <ptr>:)*
             (VALUE_DESC <ptr>:)*
```

NAME	The print name associated with this variable.
DESC	The storage descriptor for this variable (see VI.6). Each size in the SIZES field of the descriptor *must* be a code-generation-time expression.
LIFE	Lifetime attribute for this variable. It is also used to indicate the access method for the variable: local variables are allocated on the stack, static variables with the code, and parameters are passed by value.
DOT-NODE*	This is used by the XFORM phase to keep track of the value held in a variable (rather than the variable address).
OFFSET*	Offset for the current variable in its context. This is the offset in the stack (local), parameter number (parameter), or undefined for static symbols. This field is filled in by ACTREP.
LEXLEVEL*	The lexical level of this variable. This is filled in by BLOCK.
SIZE*	The static allocation for this variable (in bytes). This is filled in by ACTREP.
VAXLABEL*	The associated assembly language label for this variable, if one exists. Used only for static variables. Filled in by CODE.

TYPEDESC*	A pointer to the MIL descriptor which describes the storage layout for this variable's type descriptor at run time. Used in XFORM.
VALUE_DESC*	A pointer to the DESC node which describes the value held in this variable. Used in XFORM.

<Var> describes (fixed-size) variables. Besides describing fixed-size Ada variables, VAR nodes may be used to describe static portions of dynamic objects.

VI.6 < Desc >

```
        <Desc> →

                 DESC
                    (TYPES <Type_Desc List>:)
                    (SIZES <Stmt List>:)
                    (INIT <Stmt List>:)
                    (VARIANTS <Desc List>:)
                    (TOTALSIZE <number>)*
                    (CUMOFFSET <number list>)*
```

TYPES	(Pointers to) nodes describing each chunk of storage.
SIZES	Literals describing the number of copies (described by TYPES) of this chunk wanted.
INIT	Initial values for the corresponding chunks. Initial values may only be assigned to chunks with a size of 1. The corresponding <Stmt> field for a chunk may be NIL, indicating that no initialisation is desired.
VARIANTS	Describes the variant parts (if any) of this structure. This is currently unimplemented.
TOTALSIZE*	The number of bytes required to hold a variable with this description on the VAX. This includes all storage specified by nested DESC nodes in the TYPES attribute. This is filled in by ACTREP.
COMOFFSET*	A list of offsets which indicates how far each field is from the beginning of the storage for a variable with this description. This is filled in by ACTREP.

A DESC is used to described storage structure. The lengths of the TYPES, SIZES, and INIT lists must be the same.

VI.6.1 <Type_Desc>

```
   <Type_Desc> → <Type> | <Desc>
```

<Type_Desc> allows a field of a DESC node to reference either a TYPE or another DESC node.

VI.7 <Stmt>

```
<Stmt> → <Label> | <Block> | <Loop> | <Case> | <Return> |
         <Copy> | <Compare> | <Goto> | <Null> | <Exp> | <Exit> |
         <Raise>
```

<Stmt> nodes denote statements in the MIL. The concept of a statement is actually a misnomer, as MIL is an expression language. Every expression and statement returns a value.

VI.7.1 <Label>

```
<Label> →

        GOTOLABEL
           (NAME <string>)
           (STMT <Stmt>:)
           (BLOCK <Block>:)*
           (VAXLABEL <Label>:)*
           (HANDLER <Block>:)*
```

NAME	The print name of this label.
STMT	The statement this label is attached to. This statement will *not* appear in the STMT or HANDLERS field of a block or routine. The order of elaboration is: statement before the GOTOLABEL node, the GOTOLABEL, the statement in the STMT field of the GOTOLABEL, the statement following the GOTOLABEL node.
BLOCK*	A pointer to the closest enclosing block or routine node which contains this GOTOLABEL. Filled in by BLOCK.
VAXLABEL*	The assembly language label which marks this spot in the assembly language program. Filled in by CODE.
HANDLER*	A pointer to the closest enclosing block or routine which has an exception handler. If none, this field is NIL. Filled in by BLOCK.

GOTOLABEL nodes are used to mark places in the MIL program which may be targets for GOTO statements. The value of a GOTOLABEL is the value of its associated statement (STMT). Also see the description of the GOTO statement for restrictions on the use of GOTOs.

VI.7.2 <Block>

```
<Block> →

        BLOCK
           (LOCALS <Decl List>:)
           (BODY <Stmt List>:)
           (TYPE <Type>:)
           (ALLOC_FLAG <number>)
           (HANDLERS <Stmt List>:)
           (DYNAMIC <number>)
           (TOTALSIZE <number>)*
           (LOCALSIZE <number>)*
           (NESTEDSIZE <number>)*
           (NUMBERBLOCKINROUTINE <number>)*
           (NESTINGLEVEL <number>)*
           (FRAMESIZE <number>)*
           (MILHANDER <Label>:)*
           (BLOCKPARENT <Scope>:)*
           (HANDLERPARENT <Scope>:)*
```

LOCALS	Same as ROUTINE's LOCALS attribute.
BODY	Same as ROUTINE's BODY attribute.
TYPE	A type descriptor which indicates the value returned by this BLOCK expression.
ALLOC_FLAG	0 indicates that no dynamic allocation was done on the stack inside of this block. 1 indicates that dynamic allocation was performed and the stack must be moved back to its position before the block began execution.
HANDLERS	Same as ROUTINE's HANDLERS attribute.
DYNAMIC	1 indicates that a dynamic object is being returned on top of the stack and the stack pointer should not be altered on block exit. 0 indicates that no dynamic object is being returned on the stack. Filled in by XFORM.
TOTALSIZE*	The total amount of storage required for locals in this block and its nested blocks. Filled in by ACTREP.
LOCALSIZE*	The storage required for locals in this block only. Filled in by ACTREP.
NESTEDSIZE*	The storage required for locals in nested blocks only. Filled in by ACTREP.
NUMBERBLOCKINROUTINE*	A number assigned to each block within a procedure, starting with 1. There is no necessary relationship between this number and the nesting level of the block in the routine. Filled in by BLOCK.
NESTINGLEVEL*	Depth of block in current routine. This value is 1-based. Filled in by BLOCK.
FRAMESIZE*	The number of bytes which have already been allocated in the enclosing routine's frame, i.e., the number of bytes that been allocated before the allocation for this

	BLOCK. New locals for this BLOCK are allocated after this size. Filled in by ACTREP.
MILHANDLER*	Same as ROUTINE's MILHANDLER attribute. Filled in by CODE.
BLOCKPARENT*	Pointer to the nearest enclosing block or routine. Filled in by BLOCK.
HANDLERPARENT*	Pointer to the nearest enclosing block or routine which has an exception handler. This is the handler which would be invoked when an exception occured in the handler for this block (without any other intervening blocks). If no such scope exists, this is NIL. Filled in by BLOCK.

<Block>'s describe Bliss-style blocks. (The restrictions on the LOCALS field described in Section VI.3 apply here also.)

VI.7.3 <Loop>

```
<Loop> → <While> | <For>
```

<Loop> nodes are used to describe **while** and **for** loops.

VI.7.3.1 <While>

```
<While> →

          WHILE
             (TEST <Stmt>:)
             (DO <Stmt List>:)
             (ENDLABEL <Label>:)*
             (BLOCK <Scope>:)*
             (HANDLER <Scope>:)*
```

TEST	The test for the WHILE loop (<Stmt> must be of type *int:*). A value of 0 terminates the loop, a value of 1 continues the loop. Other values cause undefined actions.
DO	The body of the loop.
ENDLABEL*	Label (in the OBJECT language) used to mark the end of the loop for EXIT statements. Filled in by CODE.
BLOCK*	Nearest enclosing scope for this loop. Filled in by BLOCK.
HANDLER*	Nearest enclosing scope for this loop which has an exception handler (if any). Filled in by BLOCK.

WHILE nodes describe (Bliss-style) **while** loops. WHILE nodes always have an undefined return value, not 0 and not the value of any EXIT statement.

VI.7.3.2 <For>

```
<For> →

        FOR
           (INDEX <Stmt>:)
           (FROM <Stmt>:)
           (TO <Stmt>:)
           (BY <Stmt>:)
           (DO <Stmt List>:)
           (DIRECTION up | down)
           (ENDLABEL <Label>:)*
           (BLOCK <Scope>:)*
           (HANDLER <Scope>:)*
```

INDEX	The loop index (<Stmt> must be of type *loc:*).
FROM	The smaller of the initiating and terminating values. (<Stmt> must be of type *int:*).
TO	The larger of the initiating and terminating values. (<Stmt> must be of type *int:*).
BY	The increment for the index (<Stmt> must be of type *int:*).
DO	The body of the loop.
DIRECTION	This attributes desrcibes whether the loop should run from the low value to the high value, or from the high value to the low value. This has *no* impact on the sign of the BY expression.
ENDLABEL*	Label (in the OBJECT language) used to denote the end of the loop. Filled in by CODE.
BLOCK*	Nearest enclosing scope for this loop. Filled in by BLOCK.
HANDLER*	Nearest enclosing scope with has an exception handler, if any. Filled in by BLOCK.

FOR nodes describe Algol 68-style **for** loops. The value returned by a FOR loop is undefined. Note that no checks are made between the DIRECTION and BY attributes, i.e., it is possible to specify a *down* direction and a BY value of +1, though it would (probably) cause an infinite loop. Each of the INDEX, FROM, TO, and BY expressions are evaluated exactly once.

VI.7.4 <Exit>

```
<Exit> →

        EXIT
           (LOOP <Loop>:)
           (BLOCK <Scope>:)*
           (HANDLER <Scope>:)*
```

LOOP	The Mil LOOP to be exited.
BLOCK*	Nearest enclosing scope which contains this EXIT node. Filled in by BLOCK.
HANDLER*	Nearest enclosing scope, if any, which contains this EXIT node and has an exception handler. Filled in by BLOCK.

MIL EXIT nodes return undefined values. An EXIT must refer to a LOOP which encloses it and is statically nested within the same routine.

VI.7.5 <Case>

```
        <Case> →

                CASE
                   (DESIG <Stmt>:)
                   (CHOICES <Choice List>:)
```

DESIG	The expression being **case**'d (<Stmt> must be of type *loc:* or *int:*).
CHOICES	The arms of the **case** statement.

CASE nodes represent Bliss-style **case** statements. Their return value is undefined. CASE expressions are *not* implemented.

VI.7.5.1 <Choice>

```
   <Choice> → <Case_Range> | <Otherwise>
```

VI.7.5.2 <Case_Range>

```
        <Case_Range> →

                RANGE
                   (RANGE <Stmt>: <Stmt>:)
                   (BODY <Stmt List>:)
```

RANGE	Range of values for this arm of the CASE (Each <Stmt> must be of the same type as that in the DESIG field of the CASE node (VI.7.5)).
BODY	The statements comprising the code for this arm.

The return value for a RANGE expression is undefined. RANGE nodes are not implemented.

CHOICE nodes represent **case** choice ranges.

VI.7.5.3 <Otherwise>

```
        <Otherwise> →

                OTHERWISE
                   (BODY <Stmt List>:)
```

OTHERWISE nodes are for the **else** clause in **case** statements. Their return value is not defined. OTHERWISE nodes are not implemented.

VI.7.6 <Return>

```
        <Return> →

                RETURN
                   (VALUE <Stmt>:)
```

VALUE	The value to be returned.

RETURN is standard routine return and allows the return of a single unit value.

VI.7.7 <Copy>

```
        <Copy> →

                COPY
                   (DEST <Stmt>:)
                   (SOURCE <Stmt>:)
                   (SIZE <Stmt>:)
```

DEST	The destination of the COPY (<Stmt> must be of type *loc:*).
SOURCE	The source of the COPY (<Stmt> must be of type *loc:*).
SIZE	An expression denoting the number of machine units to copy (it is intended that this expression will access a location which will contain the size of the object at run time).

COPY describes a (block) bit copy operation[16].

VI.7.8 <Compare>

[16] In the current VAX implementation, the block copy works for sizes up to 2^{16}-1 bytes.

```
<Compare> →

            COMPARE
               (OP1 <Stmt>:)
               (OP2 <Stmt>:)
               (SIZE <Stmt>:)
               (TYPE bool:)
               (DYNAMIC <number>)
```

OP1	First operand to COMPARE.
OP2	Second operand.
SIZE	The number of machine units to compare (see note about SIZE field in description of COPY).
TYPE	Always *bool:*.
DYNAMIC	Either 0 or 1. See DYNAMIC attribute of BLOCK.

COMPARE is used to allow efficient implementation of the ADA equality operation. It implements a block compare and returns the integer value 1 if the two blocks of storage are equal, 0 if they are different[17]. No order of operand evaluation is specified.

VI.7.9 <Goto>

```
<Goto> →

            GOTO
               (TARGET <Label>:)
               (BLOCK <Scope>:)*
               (HANDLER <Scope>:)*
```

TARGET	The MIL GOTOLABEL node where execution should continue.
BLOCK*	Nearest enclosing scope which contains this GOTO node. Filled in by BLOCK.
HANDLER*	Nearest enclosing scope, if any, which contains this GOTO node and has an exception handler. Filled in by BLOCK.

MIL GOTO nodes return undefined values. The target of GOTOs may be in any block which shares a common block ancestor (up to the routine level) with the block containing the GOTO statement, and where no dynamic allocation is done by any block between the common ancestor block and the block containing the GOTO target. The target of the GOTO may also be in the enclosing routine of the GOTO node. Targets of GOTOs may *not* be in a different routine than the GOTO node. In essence,

[17] In the VAX implementation, the block compare is limited to objects of 2^{16}-1 bytes or less.

this means that GOTO targets cannot be in a context which requires the block's declaration elaboration to alter the stack from its height at the place of the GOTO node.

VI.7.10 <Raise>

```
        <Raise> →

                RAISE
                        (VALUE <Stmt>:)
```

VALUE	An integer expression which specifies the exception to be raised.

<Raise> nodes causes an exception to be raised explictly. The particular exception to be raised is specified by the VALUE attribute. The value of a RAISE statement is undefined.

VI.7.11 <Null>

```
        <Null> →

                NULL
```

<Null> nodes represent the **null** ADA statement. They have no side effects. They return a value of 0 if used as an expression.

VI.8 < Exp >

```
        <Exp> → <If> | <Call> | <Opr> | <Subscript> |
                <Address> | <Access> | <Literal>
```

<Exp> nodes denote (possibly value-returning) expressions in MIL.

VI.8.1 <If>

```
        <If> →

                IF
                   (COND <Stmt>:)
                   (THEN <Stmt List>:)
                   (ELSE <Stmt List>:)
                   (TYPE <Type>:)
                   (DYNAMIC <number>)
```

COND	The conditional test in the IF (<Stmt> must be of type *int:*). 1 indicates that the THEN clause should be elaborated, 0 indicates that the ELSE clause should be elaborated. Other values cause undefined actions.

THEN	The THEN clause of the IF.
ELSE	The ELSE clause.
TYPE	The type of this expression.
DYNAMIC	See DYNAMIC attribute of BLOCK.

IF nodes describe conditional expressions. The type of an IF is the type of its clauses (which must have the same type). Note that a list of statements is allowed in each clause. No elaboration is done of a clause which is not selected. The ELSE clause may be omitted, but the TYPE of the expression is required to be NONE (i.e., the THEN clause may not return a value). The value returned by an IF expression is the value returned by the selected arm.

VI.8.2 <Call>

```
<Call> →

        CALL
           (ROUTINE <Routine>:)
           (ACTUALS <Stmt List>:)
           (TYPE <Type>:)
           (DYNAMIC <number>)
```

ROUTINE	The routine being invoked.
ACTUALS	The actual parameters to be passed. Each actual must be of single unit size. Each actual is elaborated as specified by the LINKAGE attribute of the called routine.
TYPE	The return type of this call (may be obtained simply by looking at the TYPE field of ROUTINE).
DYNAMIC	See the DYNAMIC attribute of BLOCK.

CALL nodes denote routine invocations. The order of elaboration of the parameters depends on the linkage conventions of the ROUTINE. The currently implemented linkage conventions are:

C	Imitate the conventions used by the C compiler on VAX/Unix. Parameters are elaborated in *reverse* order and the routine is invoked with a CALLS instruction.
C_Dynamic	Linkage conventions used when calling a C routine, but Ada_Dynamic conventions for returning from the routine. This is used when a C run-time routine returns a dynamically-allocated object on the stack.
Ada	Linkage convention used by the Charrette compiler for simple Ada programs. See the run-time system description for these specifications. Parameters are evaluated in a left to right order and are pushed backwards on the stack. To a C program, it would appear that the parameter list was reversed. Linkage is by the CALLS instruction.

Ada_Dynamic	Linkage conventions used by the Charrette compiler for Ada routines whch return composite values on the stack. See the run-time system description for these specifications. This is the same as Ada linkage except the stack is not popped by the RET instruction for routine exit.
Algol68	Linkage conventions for Algol-68 used on VAX/Unix. As we do not know what they are, they are not implemented. They are included because the VAX/Unix debugger system claims to understand this convention.
Asm	Linkage conventions for assembler programs. See the run-time system description for these specifications. This is currently the same as C linkage.
Allocator	Linkage conventions used for calling the Ada run-time system storage allocator. See the run-time system description for these specifications.

There is no requirement that the number of actual parameters equal the number of formal parameters specified by the ROUTINE. If the linkage convention explicitly passes the number of actuals in a call, e.g., the first operand of a CALLS instruction, this number is gotten from the number of ACTUALS in the CALL node, not the number of FORMALS in the ROUTINE node.

VI.8.3 <Opr>

```
<Opr> →

        OPR
           (OP <Op>)
           (LHS <Stmt>:)
           (RHS <Stmt>:)
           (TYPE <Type>:)
           (CHECK <Stmt>: <Stmt>:)
           (DYNAMIC <number>)
```

OP	The operation to perform.
LHS	The left-hand-side operand (absent for some environment enquiries).
RHS	The right-hand-side operand (absent for unary operators).
TYPE	The type of this expression (depends on operator (<Op>) and type of operand(s)).
CHECK	Two expressions which indicate the range that the computed value must meet. This check is always made *after* the two operands are evaluated. With the exception of the assignment operations, this check is made after the operation is done. In the case of an assignment operator, the check is done before the assignment. An exception is raised if the check fails.
DYNAMIC	See the DYNAMIC attribute of the BLOCK node.

OPR nodes denote a large class of operations in MIL. See the following sections for details on each operator.

VI.8.4 <Op>

```
<Op> → <Plus> | <Minus> | <Relational> | -- | * | / | ** | rem |
       and | or | xor | not | <Dot> | <Assign> |
       <Cast> | <Environment>
```

The following paragraphs describe the effect and functionality of each <Op>. Other desired operators may be added easily as necessary.

VI.8.4.1 <Plus> and <Minus>

```
<Plus> → plus_int_int | plus_int_loc | plus_loc_int

<Minus> → minus_int_int | minus_int_loc | minus_loc_int |
          minus_loc_loc
```

<Plus> and <Minus> are used to denote addition and subtraction. Their functionalities are defined as follows:

```
plus_int_int:     INT x INT → INT
plus_int_loc:     INT x LOC → LOC
plus_loc_int:     LOC x INT → LOC

minus_int_int:    INT x INT → INT
minus_int_loc:    INT x LOC → LOC
minus_loc_int:    LOC x INT → LOC
minus_loc_loc:    LOC x LOC → LOC
```

Operands of type LOC and INT may be used for address calculation. *Minus_loc_loc* is defined for use in offset calculation. Integer arithmetic may cause hardware overflows, which are trapped as exceptions and processed. Location arithmetic (any <plus> or <minus> operation with a LOC operand) will not cause an overflow. If the result is too large for the machine, the most significant bits will be discarded.

VI.8.4.2 <Relational>

```
<Relational> → lss_int | leq_int |
               gtr_int | geq_int |
               eql_int | neq_int |
               eql_loc | neq_loc
```

<Relational> operators describe the standard single unit relational operations. The return type of a <Relational> operation is always *bool*: where 0 indicates false and 1 true. These operators are *not* meant to be used to perform the general ADA equality operation (see Section VI.7.8 for a bit comparison operator).

All *_int* operators have the following functionality:

```
INT x INT → INT
```

while *eql_loc* and *neq_loc* have functionality

```
LOC x LOC → INT
```

VI.8.4.3 --

-- is unary integer negation:

```
--:     INT → INT
```

The RHS field for -- is irrelevant. Overflow may occur on two's complement machines.

VI.8.4.4 *, /, rem and **

* is integer multiplication, / is integer division, rem is integer remainder and ** is integer exponentiation. Overflow may occur on some machines. The functionality for all three is:

```
INT x INT → INT
```

Note that these operations are not defined on LOC's.

VI.8.4.5 *and*, *or* and *xor*

These are the logical operators with the value 0 representing false and 1 true. Their functionalities are

```
INT x INT → INT
```

VI.8.4.6 *not*

Not is unary complement:

```
not:     INT → INT
```

The RHS field for *not* is irrelevant. Note that NOT 0 = 1 and NOT 1 = 0.

VI.8.4.7 <Dot>

```
<Dot> → dot_loc | dot_int
```

<Dot> is the indirection operator:

```
dot_loc:     LOC → LOC
dot_int:     LOC → INT
```

The return type of the <Dot> operator must be defined in the operator (i.e., what are we indirecting to).

VI.8.4.8 <Assign>

<Assign> → *assign_loc* | *assign_int* | *rev_assign_loc*

<Assign> is meant to denote single unit assignment/copy. (For a block copy operation, see VI.7.7.)

assign_int:	LOC x INT → INT
assign_loc:	LOC x LOC → LOC
rev_assign_loc:	LOC x LOC → LOC

<Assign> is *not* intended for general representation of the ADA assignment operation (although it might be used for assignment of integers, characters, etc.). This operator is intended for use in assignment of internal entities (e.g., array descriptors, range constraints, etc.). Both sides are evaluated before the assignment is done. Normally, the LHS describes the target for the asignment and the RHS the value to be assigned. In the case of the reverse assignment operator, *rev_assign_loc*, the target of the assignment is the RHS expression, not the left hand side expression. Any specified checking is done *before* the assignment is done.

VI.8.4.9 <Cast>

<Cast> → *cast_loc_int* | *cast_int_int* | *cast_int_loc*

<Cast> is used for coercing a datum of one type into a datum of another type.

cast_loc_int:	LOC → INT
cast_int_int:	INT → INT
cast_int_loc:	INT → LOC

<Cast> is not intended to implement Ada type conversion, only to provide a limited facility for changing between location and integer data, and between integer data of different sizes.

VI.8.4.10 <Environment>

<Environment> → *radix* | *size* | *storage_unit* | *excep_val*

<Environment> is used for determining values in the run-time environment. All of the environment inquiries return integer values.

RADIX	The radix of the machine's arithmetic. Currently this value is 2.
SIZE	The number of bits in the expression in the LHS.
STORAGE_UNIT	The number of bits in the basic storage unit of the machine. Currently this value is 8.

EXCEP_VAL	Evaluates to the value of the current (pending) exception. If no exception is pending, this value is undefined.

<Environment>'s are intended to implement some of Ada's attribute enquiries.

VI.8.5 <Subscript>

```
        <Subscript> →

                [] 
                   (BASE <Stmt>:)
                   (INDICES <Stmt List>:)
                   (TYPE loc:)
                   (BOUNDS <Bounds>:)
                   (DYNAMIC <number>)
                   (INDIRECT_TYPE <Type>:)*
```

BASE	The base address of the dope vector for the array being subscripted (<Stmt> must be of type *loc:*).
INDICES	The indices of the subscript (each <Stmt> must be of type *int:*).
TYPE	The type returned by [ ] (must be *loc:*).
BOUNDS	The bounds information for this array (VI.8.5.1). This is currently unused and unimplemented.
DYNAMIC	See the DYNAMIC attribute for BLOCKs.
INDIRECT_TYPE*	Attribute used by XFORM to indicate the contents of the address specified by this operator.

[] is used to allow efficient subscripting (particularly on the VAX). The BOUNDS field (bounds descriptor) is passed to SUBSCRIPT so that it may take advantage of information contained therein (e.g., static bounds, fixed size of elements, etc.). BOUNDS is currently unimplemented.

The fact that the base address of the dope vector for the array is passed to SUBSCRIPT implies that the producer of MIL and the code generator must agree on the structure of array dope vectors.

The address calculations for a subscript, like all address calculations, will not generate an overflow, although evaluation of the indices themselves may cause an overflow. Bounds checking for the array is generated automatically and cannot be disabled.

VI.8.5.1 <Bounds>

```
        <Bounds> →

                 BOUNDS
                    (LBS <Stmt List>:)
                    (UBS <Stmt List>:)
                    (ELT_SIZE <Stmt>:)
```

LBS	*int:* expressions giving the lower bounds for each dimension of the array.
UBS	*int:* expressions giving the upper bounds for each dimension of the array.
ELT_SIZE	The size (in machine units) of each element of the array (<Stmt> must be of type *int:*).

BOUNDS nodes represent bounds and element type information for arrays. They are currently unimplemented.

VI.8.6 <Address>

```
        <Address> →

                 GTADDRESS
                    (WHERE <Var>:)
                    (TYPE loc:)
                    (DYNAMIC <number>)
```

WHERE	The variable whose address is desired.
TYPE	The type returned by GTADDRESS (always *loc:*).
DYNAMIC	See the DYNAMIC attribute of BLOCK.

GTADDRESS is used to obtain the address of a variable. Its use is analogous to the occurrence of a variable name in Bliss which denotes the *address* of the variable.

VI.8.7 <Access>

```
        <Access> →

                 ACCESS
                    (BASE <Stmt>:)
                    (FIELD <Integer>)
                    (DESC <Desc>:)
                    (VAR_NUM <integer>)
                    (TYPE loc:)
                    (DYNAMIC <number>)
```

BASE	The base address of the structure (the type of <Stmt> must be *loc:*).
FIELD	The ordinal number of the field to be accessed (1-based).
DESC	The storage descriptor describing the structure.
VAR_NUM	The index of the VARIANT field (in the DESC node) to use (1-based). This is currently unimplemented.
TYPE	The type of value returned by ACCESS (always *loc:*).
DYNAMIC	See the DYNAMIC attribute of BLOCK.

ACCESS is used to access a field in a MIL structure (i.e., storage layed out for a VAR node). If the field to be accessed lies in the variant part of the structure then the VAR_NUM field tells which variant descriptor to use (see VARIANT field in Section VI.6).

VI.8.8 <Literal>

```
        <Literal> →

                LITERAL
                   (TYPE <Type>:)
                   (VALUE <Value>)
                   (DYNAMIC <Number>)
```

TYPE	Type of this LITERAL.
VALUE	The actual value. Note that all LITERAL values are single unit quantities.
DYNAMIC	See DYNAMIC attribute for BLOCK nodes.

LITERAL nodes are used to represent literals (constants) in the MIL program.

The following LITERAL nodes denote the constants 0 and 1 (they were referenced in Section VI.4).

```
zero:     LITERAL
                  (TYPE int:)
                  (VALUE 0)

one:      LITERAL
                  (TYPE int:)
                  (VALUE 1)
```

VI.8.8.1 <Value>

```
        <Value> → <integer> | <Plit>
```

<Value> nodes describe values similar to those used in a Bliss **bind**.

VI.8.8.2 <Plit>

```
<Plit> →

          PLIT
             (TRIPLE <Value List>)
             (VAXLABEL <Label>:)*
```

TRIPLE	The data pointed to (as in a Bliss **plit**).
VAXLABEL*	Assembly language label which starts storage for these literals. Filled in by CODE.

A PLIT node creates a pointer to a literal. Restricted forms of PLIT nodes are currently implemented. The TRIPLE value must be a list of integers, strings, and plit nodes.

VII. Bibliography

[1] W. A. Barrett and J. D. Couch.
Compiler Construction: Theory and Practice.
Science Research Associates, 1979.

[2] G. Birtwistle, L. Enderin. M. Ohlin, and J. Palme.
DEC-System 10 Simula Language Handbook.
Rapportcentralen, FOA 1, S-104, Stockholm 80 Sweden, 1976.
Report C8398.

[3] B. M. Brosgol. J.M. Newcomer, D.A. Lamb, D. Levine, M. S. Van Deusen, and W.A. Wulf.
$TCOL_{Ada}$: 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.

[4] B. M. Brosgol.
CWVM: The "Middle End" of the PQCC Ada Compiler.
In *Symposium on the Ada Programming Language.* ACM, Boston, December, 1980.

[5] Digital Equipment Corporation.
VAX-11 Architecture Handbook.
1979.

[6] Digital Equipment Corporation.
VAX-11 Software Handbook.
1977.

[7] Digital Equipment Corporation.
VAX-11/780 Hardware Handbook.
1977.

[8] H. Ganzinger and K. Ripken.
Operator Identification in Ada: Formal Specification, Complexity, and Concrete Implementation.
Sigplan Notices 15(2):30-42, February, 1980.

[9] D. Gries.
Compiler Construction for Digital Computers.
John Wiley and Sons, Inc., 1971.

[10] Hilfinger, P. N.
Discriminant Constraints in Ada, LIR.008.
Available on-line at ARPA-net site ISIE as <TNE-ARCHIVE> LIR.008.

[11] Hilfinger, P. N.
Implementation of Composite Objects with Components of Varying Size.
Unpublished memorandum.

[12] A. Hisgen, D. A. Lamb, J. Rosenberg, M. Sherman.
A Runtime Representation for Ada Variables and Types.
In *Symposium on the Ada Programming Language.* ACM, Boston, December, 1980.

[13] R.C. Holt and D.B. Wortman.
A Model for Implementing Euclid Modules and Type Templates.
In *SIGPLAN Conference on Compiler Construction*, pages 8-12. ACM, Denver, 1979.

[14] *Reference Manual for the Ada Programming Language*
Honeywell, Inc., 1980.

[15] J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner, B.A. Wichmann.
Reference Manual for the Ada Programming Language.
SIGPLAN Notices 14(6):1, June, 1979.

[16] J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner, B.A. Wichmann.
Rationale for the Design of the Ada Programming Language.
SIGPLAN Notices 14(6):1, June, 1979.

[17] J.D. Ichbiah, B. Krieg-Brueckner, B.A. Wichmann, H.F. Ledgard, J.C. Heliard, J.R. Abrial, J.G.P. Barnes, M. Woodger, O. Roubine, P.N. Hilfinger, R. Firth.
Reference Manual for the Ada Programming Language
The revised reference manual, July 1980 edition, Honeywell, Inc., and Cii-Honeywell Bull, 1980.

[18] J. M. Janas.
A Comment on "Operator Identification in Ada" by Ganzinger and Ripken.
Sigplan Notices 15(9):39-43, September, 1980.

[19] D. A. Lamb.
A Pattern-Driven Peephole Optimiser.
Technical Report CMU-CS-80-141, Carnegie-Mellon University, Computer Science Department, August, 1980.

[20] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek.
Report on the Programming Language Euclid.
SIGPLAN Notices 12(2):1-79, February, 1977.

[21] B.W. Leverett, R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz, W.A. Wulf.
An Overview of the Production Quality Compiler-Compiler Project.
Technical Report CMU-CS-79-105, Carnegie-Mellon University, Computer Science Department, February, 1979.

[22] T. Pennello, F. DeRemer, and R. Meyers.
A Simplified Operator Identification Scheme for Ada.
Sigplan Notices 15(7&8):82-87, July-August, 1980.

[23] G. Persch, G. Winterstein, M. Dausmann, and S. Drossopoulou.
Overloading in Ada.
Berich 23/79, Institut fur Informatik II, Universitat Karlsruhe, November, 1979.

[24] B. Randell and L. J. Russell.
Algol 60 Implementation.
Academic Press, 1964.

[25] J. Rosenberg, D. A. Lamb, A. Hisgen, and M. S. Sherman.
The Charrette Ada Compiler.
In *Symposium on the Ada Programming Language*. ACM, Boston, December, 1980.

[26] M.S. Sherman and M. Borkan.
A Flexible Semantic Analyzer for Ada.
In *Symposium on the Ada Programming Language*. ACM, Boston, December, 1980.

[27] M.S. Sherman, A. Hisgen, D. A. Lamb, and J. Rosenberg.
An Ada Code Generator for VAX 11/780 with Unix.
In *Symposium on the Ada Programming Language*. ACM, Boston, December, 1980.

[28] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard.
Simula BEGIN.
Van Nostrand Reinhold, 1973.

[29] P.J.L. Wallis and B.W. Silverman.
Efficient Implementation of the Ada Overloading Rules.
School of Mathematics, University of Bath, Claverton Down, Bath BA2 7AY, UK.

[30] W.A. Wulf, D.B. Russell, and A.N. Habermann.
BLISS: a Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

[31] W. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke.
The Design of an Optimizing Compiler.
American-Elsevier, 1975.

[32] W.A. Wulf, M. Shaw, P.N. Hilfinger, and L. Flon.
Fundamental Structures of Computer Science.
Addison-Wesley, 1981.

[33] Wulf, W. A.
Private communication.