CMU-CS-79-128

TCOL$_{Ada}$: Revised Report on An Intermediate Representation for the DOD Standard Programming Language

20 June 1979

Joseph M. Newcomer
David Alex Lamb
Bruce W. Leverett
David Levine**
Andrew H. Reiner
Michael Tighe**
William A. Wulf

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

**Intermetrics, Inc., Cambridge, MA 02138

This research was sponsored by the Defense Advanced Research Projects Agency (DOD). The Carnegie-Mellon contract is monitored by the Air Force Avionics Laboratory.

Table of Contents

Table of Figures

1

Preface to the 20 June edition

Because of tight publication deadlines, primarily the need to circulate a draft of this specification widely by the end of June, some sections were not completed. We expect these sections to be completed in the final draft. Many sections contain no prose because there is nothing in the Ada manual which applies to $TCOL_{Ada}$. For completeness, these sections are left in this manual.

We solicit feedback on this edition of the document. Comments, questions, and suggestions may be sent to:

Joseph M. Newcomer
Computer Science Department
Carnegie-Mellon University
5000 Forbes Avenue
Pittsburgh, Pa. 15213

or via the ArpaNet to:

Newcomer@CMU-10A

Later editions may be obtained by writing to the above U.S. Mail address, or by sending a request via the ArpaNet. This document is also available in machine-readable form suitable for printing on line printers, DECwriters^{tm}, Diablo^{tm} or equivalent devices, and as general ASCII text for printing on other devices[1]. The machine-readable source, for the SCRIBE document production system, is also available. Direct inquiries to the above addresses.

[1]The primary difference among these devices is how underlining and overstriking are done; such features enhance the readability of the output when they are available.

1. Format of this document

The document is presented in several sections. The introductory and overview prose is in numbered chapters; chapter 2 is the introduction to TCOL; chapter 3 is a brief overview of the language used to express TCOL.

The bulk of the document is given with chapters and sections with the prefix "Ada" and is keyed to the Ada Reference Manual [2]. If a section number is given with a letter suffix, e.g., "Ada-5.6.c", then that represents a finer breakdown than given in the Ada reference manual for a particular section, e.g., section 5.6. Several appendices summarize the information distributed throughout the manual. A comprehensive index and a bibliography are included.

Editorial comment, annotations, explanations, and other prose not related directly to the content of the document, but which may aid the reader's understanding either of the document or the motivations of the authors in making particular design choices is shown like this.

2. Introduction

This document describes $TCOL_{Ada}$, an intermediate representation for programs written in Ada. $TCOL_{Ada}$ is intended to be a uniform, machine-independent representation of Ada programs suitable for further processing by machine-dependent compiler modules. It is intended that the $TCOL_{Ada}$ produced by a parser/semantic analyzer be usable by many different implementations of Ada compilers for many different machines.

This document uses the term "intermediate representation" to denote languages suitable for representing source programs in the innards of a compiler. $TCOL_{Ada}$, one such intermediate representation for an Ada compiler, is described here.

TCOL is the generic name given to a set of language-specific TCOL instantiations such as $TCOL_{Ada}$, $TCOL_{Pascal}$, and $TCOL_{Bliss}$. All of the specific TCOLs are very similar; they differ in that each contains constructs for handling features unique to its language. For instance, $TCOL_{Fortran}$ would contain a construct for the DO statement, $TCOL_{Bliss}$ would have the ability to represent byte pointers, and so on.

TCOL was originally developed as tool for use in the Production Quality Compiler Compiler (PQCC) project at Carnegie-Mellon University [3]. PQCC is investigating techniques for automating compiler construction. A Production Quality Compiler (PQC) produced by this technology is expected to be as efficient as the best hand-built compilers.

A PQC is phase-structured; it is composed of a linear sequence of phases that each perform some task in the code generation process. Dialects of TCOL provide communication among the various phases. For developmental purposes, it is important that the TCOL be human readable (i.e., have an ASCII representation). It is also important that TCOL primarily represent the semantics of the language; this allows the compiler to maximize the scope and magnitude of its optimizations. TCOL was designed so that its internal representation can be very efficient; a production version of a compiler would not need to write the text files unless requested to do so.

The language used to express $TCOL_{Ada}$ is called "LG", and is described briefly in chapter 3.

It is important to understand that $TCOL_{Ada}$ serves two purposes: one is to specify the intermediate representation of Ada programs, and the other is to make this intermediate representation visible to people and other programs. Although the TCOL representations shown here look complex, in fact they represent exactly the information that an equivalent internal form would possess. LG was designed to be a readable form of the conventional internal form of such complex structures, so that in particular one is not forced to read octal dumps to determine the source of an error. Within a research environment, it enabled separate phases of the compiler to be built independently, because each phase would read and write TCOL text files; in practice, a compiler could pass information from phase to phase through memory, exactly as conventional compilers do today.

The advantages of using a TCOL representation for Ada programs are numerous:

- A tree-structured intermediate representation is more suitable for program manipulation (e.g., optimization) than most other forms. Ada is a language in which there are many opportunities for program manipulation of various forms for optimization purposes.

- The ability to read and write an external form of $TCOL_{Ada}$ allows for more flexibility in designing and building compilers.

- Separate development of compiler phases is possible, and such development can proceed on different machines; for example, a complete parser/semantic analyzer may be developed, and its output could still be machine-independent. Machine-dependent code generators could then be produced independently, with varying degrees of sophistication. A complete new system would not have to be brought up for each new machine.

- It will provide a medium of communication among the various groups constructing Ada compilers. Implementors will speak the same "language" when discussing how their compilers work.

3. LG

LG is fully described in [4]. A brief overview is given here. In addition to the LG notation, a set of tools for reading, writing, and manipulating LG files exists, and a set of tools for managing systems which use LG has been developed.

LG is a notation for expressing, in the form of readable text, the internal data structures for a compiler or other complex data manipulation system. It was designed to meet the following requirements:

- The notation should be able to represent an arbitrary directed graph with many links, including cyclic links.
- The notation should be able to represent information independently of its implementation, e.g., representing a sequence of data which may be stored as a list, a set, a vector, etc.
- The notation should be transformable to an efficient representation, e.g. a highly packed bit representation with single bits for booleans, small fields for small values, etc.
- The notation should permit two phases which communicate by writing to an intermediate file to be combined and communicate directly by passing the data structures in memory.
- The implementation of a system which uses LG should pass information it does not understand idempotently through the phase, so that information is not lost.

These goals were driven by the desire to produce a system which was comfortable and friendly for developing a system as a research system, and yet suitable for building a true production version of the same system without requiring a complete recoding.

We will first give an example in LG, and then explain the details of the notation.

```
17: OBJECT
      (NAME BALL)
      (COLOR YELLOW)

23: ACTOR
      (NAME JACK)
      (AGE 6)

31: RELATION
      (NAME PLAYS-WITH)
      (WHAT 23:)
      (TOWHAT 17:)
```

Figure 3-1: LG example

This example was chosen because it has nothing whatever to do with compilers. It is therefore possible to concentrate on what the notation says without worrying about what we must say to describe a compiler data structure.

This shows that there exist things called OBJECTs that have names and colors, ACTORs that have names and ages, and RELATIONs for connecting actors to objects (or possibly objects to actors), which have names and directed arcs WHAT and TOWHAT. Attribute names such as "NAME", "AGE", "WHAT", and "TOWHAT" are not interpreted by the LG support system -- any other identifiers could have been used equally well. Moreover, the NAME fields in the three types of nodes, OBJECTs, ACTORs, and RELATIONs, are not necessarily related to each other, or confused or connected with each other in any way by the LG system. Thus LG could be the external representation of a conventional record structure, as provided by languages like PASCAL.

3.1 Primitive data types

The primitive types for the attribute values are:

integer represented externally by a string of digits, or by a symbolic name;

label — represented by an octal number followed by a colon (forward references are handled correctly).

identifier — represented by a string of letters, digits, and even some punctuation marks;

string — quoted strings of arbitrary characters;

sequence — sequences of values (separated by blanks) of any of the above types, possibly with various types intermixed.

Values of the *identifier* type are represented internally by unique integers generated by the LG system; two of them can be tested for equality, but no other meaningful operations can be performed.

An LG support package provides the software necessary to work with these representations in a program. It contains:

- A definition-file generator, which takes a specification of the node types, attribute names, and allowable value types and values, and produces definition files used by the source program. These files provide the necessary access to the fields, to the node information, and to the representation. They additionally define the tables required by the input/output support.

- Input/output runtime support, which reads and writes LG files.

- Runtime utility support, which provides procedures for set and list manipulation, storage management, creation and deletion of nodes and complex values, and error handling.

Attributes of type *integer* and *identifier* frequently appear similar in the external representation. This is because of the facility for defining symbolic names for integer attribute values. Consider, for instance, the attribute COLOR, of "object" things. The user can specify that the only legitimate colors have symbolic names BLUE, RED, YELLOW, and GREEN, and can further specify which integers these four names represent. If, alternatively, the COLOR attribute had type *identifier*, then any name would be a legitimate color; two colors could be tested for equality, but no other operations (such as typical integer operations) would be meaningful.

Attribute names and symbolic names, like identifiers, need only conform to the very permissive LG syntax for identifiers. Since most languages (BLISS in particular) have more restricted identifier syntax, the LG facility for defining them allows them to be associated with "internal" identifiers, which are expected to obey the rules of the host language.

3.2 Composite data types

The internal representation of a sequence is defined by the user; thus, the sequence

```
(SUBNODES 17: 44: 76: 122: 5:)
```

may be stored as

an *array*: the order is preserved, and the i^{th} element of the array is the i^{th} value in the sequence;

a *set*: the order is not preserved, and duplicate entries are omitted. Insertion and retrieval are efficient;

a *list*: the order is preserved, and insertions and deletions are efficient while indexing is not (lists are doubly linked).

(All of these representations are fully supported by the LG software.)

In addition, atomic types or arrays may contain values of type *Item*. An *Item* has a value which can be any of the atomic types or composite types, and has a type-tag indicating which type the value possesses. For example, the following sequence could be stored only in an item-array, set, or list:

```
(THING-SEQUENCE "string" 17: 45 any-id)
```

Similarly, the following two nodes would require that the VALUE field be of type *Item*, and the type of the item would be determined at run time by examining a tag field.

```
17: SOMENODE
        (VALUE 44:)

23: SOMENODE
        (VALUE 5)
```

In this example, the type tag associated with the VALUE field of node 17: would indicate that the type of the VALUE field is *label*, and the type tag of the VALUE field of node 23: would indicate that the type of the value field was *Integer*. As with "union mode" or "variant record" features in many languages, this feature defeats some of the type checking that normally is done.

4. The Compiler Model

TCOL is a family of languages suitable for expressing the intermediate representation of programming languages during the compilation process. There are major variants of this family, e.g., TCOL$_{\text{BLISS}}$ which represents programs in BLISS, and TCOL$_{\text{Ada}}$ which represents programs in Ada. There could also be TCOL$_{\text{Fortran}}$, TCOL$_{\text{Pascal}}$, etc. It is assumed that the commonality of these languages is greater than their differences, so in fact there is some "core" which is actually common to all languages. Extensions can be done so that some level of the compiler could actually accept TCOL for several languages.

However, even within one TCOL there are many dialects; these represent the additional information added by the various phases of the compilation process, or in some cases, a "simpler" TCOL dialect represents the binding of certain decisions and the consequent discarding of information required to make the binding.

The compiler model, at a first approximation, is shown in figure 4-1. It consists of a Front End, which produces TCOL$_{\text{Ada[F/E]}}$, a module referred to as "CWVM"[2] which binds implementation decisions and produces TCOL$_{\text{Ada[CWVM]}}$, and a Back End which generates code, and whose output is machine code. Within each of these phases there can be several dialects of TCOL$_{\text{Ada}}$.

This document specifies TCOL$_{\text{Ada}}$ as output by the Front End, i.e., semantic analysis has been done.

It is important to realize that this is a *model* of a compiler for purposes of exposition. It is not a specification for the construction of a compiler. For example, a Front End may be done as a separate parser and semantics analyzer which communicate through files written in TCOL, or as a single phase from which the TCOL$_{\text{Ada[F/E]}}$ is produced.

The TCOL$_{\text{Ada}}$ as specified here is suitable as input to a CWVM module. A given

---

[2]For "Compiler Writer's Virtual Machine" [1].

```
+-------------+        +--------------+          +--------------+
|             |        |              |          |              |
|  Front End  |------->|     CWVM     |--------->| Back End     |
|             | TCOL_FE |             |TCOL_CWVM |              |
+-------------+        +--------------+          +--------------+
```

Figure 4-1: Ada Compiler as viewed in this document

implementation may actually incorporate the CWVM functions into the Front End, using a much richer representation internally than this specification requires. Its output would be the $TCOL_{Ada[CWVM]}$ shown in figure 4-1. However, if such a module were able to additionally produce a TCOL which satisfied the specifications of this document, it would be suitable as an Ada Front End to any other system which accepted the TCOL defined here as input. Such a decomposition is shown in figure 4-2.

```
+------------+                                        +------------+
|            |                                        |            |
| Front End 1|--------------------------------->| Back End 1 |
|            |   TCOL_FE.1                    ^      |            |
+------------+                                |      +------------+
        |                                     |
        --------> TCOL_Ada                    |
                                              |
                      +-------------+         |
                      |             |         |
TCOL_Ada ------------>| Translation |----------
                      |             |
                      +-------------+
```

Figure 4-2: Compiler decomposition with enhanced TCOL

In example 4-2, a particular implementation of a Front End produces an enhanced

TCOL for its associated Back End. This may simply include more pointers of various sorts, e.g. sibling pointers in record components, ancestor pointers in TREE_NODEs, etc., or may have other extensions which represent information the Front End has discovered and which, if the communication were in pure $TCOL_{Ada}$, the Back End would have to discover for itself. However, the Front End also puts out a subset of $TCOL_{FE.1}$ which satisfies the $TCOL_{Ada}$ specification, and a "translation" program exists which will take $TCOL_{Ada}$ and add the necessary enhancements required to achieve $TCOL_{FE.1}$. Such a compiler structure satisfies the requirements of producing and accepting $TCOL_{Ada}$.

The TCOL output by the Front End expresses a program entirely in terms of language semantics. No implementation-specific or machine-specific semantics are in the $TCOL_{Ada[F/E]}$. The TCOL output by the CWVM expresses a program in terms of machine and implementation semantics as well, e.g., addition is no longer a single operator, but the various sorts of addition supported by the target machine and which are appropriate for the source language data types are all identified.

5. The Representation Model

The representation is inspired by the notion of class and subclass from SIMULA-67. However, the limits of the LG notation require that extensions to a basic class be done by creating new "nodes" (which would be called "records" in some languages). The hierarchy used in this document is shown in figure 5-1.

This shows the hierarchical relationships among the nodes which represent declarations. The first level, consisting only of NAME_NODEs, is the "name table" of a compiler. The next level, those nodes which can be referred to by a NAME_NODE, is the "symbol table" of a compiler. The LITERAL_REP nodes which are referred to by VARBL_SYM nodes comprise the "literal table". The remaining _REP nodes (ACCESS_REP, ARRAY_REP, etc.) are extensions to the TYPE_SYM node.

In a conventional record-oriented language, these could be thought of as variants in the TYPE_SYM record. In LG, the variants are implemented as new nodes, so the discriminant on the variant is the LG node-type, which is easily determined.

The hierarchy for the nodes which represent the executable program text is shown in figure 5-2. LEAF_NODEs are an extension of TREE_NODEs, and DECLARATION_INFO provides additional information for certain types of operators.

```
TREE_NODE
   |
   +----------LEAF_NODE
   |
   +----------DECLARATION_INFO
```

Figure 5-2: Hierarchy for program tree nodes

The exact specification for LINKAGE_INFO nodes, and their relationship to other nodes in figure 5-1, is not complete.

```
NAME_NODE
    |
    |
    +----------TYPE_SYM
    |             |
    |             +-------------ACCESS_REP
    |             |
    |             +-------------ARRAY_REP
    |             |
    |             +-------------CONSTRAINT_REP
    |             |
    |             +-------------ENUMERATION_REP
    |             |
    |             +-------------RECORD_REP
    |             |
    |             +-------------SCALAR_REP
    |
    +----------EXCEPTION_SYM
    |
    +----------LABEL_SYM
    |
    +----------PRAGMA_SYM
    |
    +----------PACKAGE_SYM
    |
    +----------SUBPROGRAM_SYM
    |
    +----------TASK_SYM
    |
    +----------VARBL_SYM
                  |
                  +--------------LITERAL_REP
```

Figure 5-1: Hierarchy for names, symbols, types, etc.

6. Notation

This section deals with how TCOL nodes will be represented in this document *for purposes of exposition*. Each node will be presented in skeleton form, which will be a complete specification of the node. Usually, when a node appears in an example, only a partial node will be shown.

TCOL nodes are described as in figure 6-1. TCOL does not distinguish upper and lower case, so frequently, for purely aesthetic reasons, some TCOL examples contain lower case text. In addition, "non-terminal" symbols in the LG notation are shown highlighted, as in figure 6-1. In this example, the names "label:" and "identifier" stand for any LG label and any LG identifier.

```
label:   TREE_NODE
         (OP identifier)
```

Figure 6-1: TCOL representation of a node

To enhance readability, this document uses symbolic labels in the LG examples. Actual LG support requires octal integer labels, which present no problem when the TCOL is generated by machine.

A simple SNOBOL program exists which will do the translation when it is required. Any program which generates TCOL should use the octal labels, to eliminate the need for an extra step in the compilation process. Although the program is simple, it is slow, and it requires two passes.

An attribute value which is actually an LG label will be shown prefixed with the name of the node it points to. When it can point to several different types of nodes, the types of nodes are usually given as a comment, as shown in figure 6-2. Because expressions in TCOL can be represented by either TREE_NODEs or LEAF_NODEs, and because statements are also represented as TREE_NODEs, the special "node type" *expr* is used as a notational convenience to indicate a pointer to either a TREE_NODE, or where reasonable, a LEAF_NODE.

```
label:   TREE_NODE
         (OP call)
         (SUBNODES SUBPROGRAM_SYM-label: expr-label-sequence)

label:   TYPE_SYM
         (NAME NAME_NODE-label:)
         (REP label:)                          !To ARRAY_REP,
                                               ! RECORD_REP,
                                               !ENUMERATION_REP,
         ! ... etc.
```

Figure 6-2: Notation for labels in attributes

It is frequently inappropriate or unwieldy to give complete examples, so several forms of ellipses are used:

- In examples of Ada code, comments are frequently used to indicate "declarations" or "statements" where the exact contents are irrelevant.

- In examples of Ada code, where specific expressions or statements are to be shown in their relation to the TCOL tree, arbitrary groups of statements are designated by s_i and arbitrary expressions by e_i. The TCOL expansion of these statements is not shown in the TCOL representation.

  while e0 loop s1 end loop;

  ```
  label:  TREE_NODE
          (OP while)
          (SUBNODES e0: s1:)
  ```

- Attributes which are not relevant to the example are usually omitted; for example, the SOURCE attribute which is present in every node hardly ever appears in the examples; the NAME attribute in VARBL_SYM nodes and some others, which is simply a reference to the print name, is frequently omitted.

- Within an attribute, which can consist of a sequence of LG items, a sequence of dots indicates that several such items may precede or follow the item shown, e.g.

```
(SUBNODES ... something: ...)
```

The comment "etc." is used frequently in node descriptions to indicated that some attributes are not shown.

- When a reference is made to an expression which has a numeric value, and that value is a literal, a label with the literal name is given, but no further description is given, as shown in figure 6-3.

```
sometree:        TREE_NODE
                 (OP +)
                 (SUBNODES ... one: ...)
```

will imply the expansion of "one:" which is:

```
one:     LEAF_NODE
         (OP leaf)
         (SUBNODES lit-1:)

lit-1:   VARBL_SYM
         (CONSTANT COMPILE)
         (INITIALIZE litval-1:)

litval-1:        LITERAL_REP
                 (VALUE 1)
```

Figure 6-3: Simplified representation of literals

7. Node types

ACCESS_REP	Describes the properties of an access type variable.
ARRAY_REP	Describes the properties of an array.
CONSTRAINT_REP	Describes the constraints of a type, subtype, or derived type.
DECLARATION_INFO	Describes the declarations to be processed for a subprogram, module, block, etc.
ENUMERATION_REP	Describes the properties of an enumeration type.
EXCEPTION_SYM	Describes an exception, either predefined or user-defined.
GENERIC_INFO	Links together the instances of a generic subprogram.
LABEL_SYM	Describes the properties of a program <<label>>.
LEAF_NODE	A leaf node in the program tree, e.g., nodes representing variables or constants.
LINKAGE_INFO	A node which contains the details of the parameter passing mechanism for a subprogram.
LITERAL_REP	A node which holds the value of a literal. LITERAL_REP nodes may be pointed at *only* by VARBL_SYM nodes.
NAME_NODE	Holds the source language name; either an identifier or a literal.
PACKAGE_SYM	Describes the properties of a package.
PRAGMA_SYM	Describes a language pragma.
RECORD_REP	Describes the properties of a record.
SCALAR_REP	Describes the properties of scalar types for fixed, float, integer and boolean types.
SUBPROGRAM_SYM	Describes the properties of a procedure, value-returning procedure, function, or entry.
TASK_SYM	Describes the properties of a task.
TREE_NODE	A interior node in the "program tree", e.g., an operator node in an arithmetic expression.

TYPE_SYM	Describes the properties of a type, derived type or subtype
VARBL_SYM	Describes the properties of a variable, constant, formal parameter, or record component.

7.1 The SOURCE attribute

All TCOL nodes possess a SOURCE attribute. The SOURCE attribute is a string which, when given to a suitable program for the machine and operating system, will locate the source character from which the node was created (in the case of SYMBOL nodes, for example, this would be the first character of the lexeme in a declaration).

For example, on TOPS-10, a suitable string for a sequence-numbered file would be "FILE.EXT;line/page{char}", e.g., "MYPROG.ADA;00100/5{47}"; without sequence numbers, the "line" part would be the count of lines within the page, e.g., "MYPROG.ADA;1/5{47}".

This information is used to report error conditions during other phases of the compiler. In addition, this information may be used by the code generator and passed to a debugging environment so that errors, debug printout, etc. may be related back to the source program. If clever encodings are appropriate for representing this information, these decisions belong elsewhere than the Front End; the Front End should deliver a straightforward representation of the location in a form which is easily human-readable.

The exact form of the SOURCE attribute in the tree is implementation-dependent, but must be powerful enough to allow access to the source file in the environment of the system. This means that the representation must be appropriately chosen for the system.

Appendix Ada: TCOL for Ada

Ada-1. Introduction

Ada-2. Lexical elements

Ada-2.1 Character set

Ada-2.2 Lexical Units and Spacing Conventions

Ada-2.3 Identifiers

Identifiers are represented by NAME_NODEs; see section Ada-4.1.

Ada-2.4 Numbers

Numbers are represented by VARBL_SYM nodes which in turn refer to LITERAL_REP nodes.

The exact representation for real values is discussed in section Ada-3.5.5.

Ada-2.5 Character Strings

Ada-2.6 Comments

Ada-2.7 Pragmas

A language pragma is described by a PRAGMA_SYM node.

```
label:    PRAGMA_SYM
          (NAME NAME_NODE-label:)
          (ARGS label-sequence)
```

Figure Ada-2-1: PRAGMA_SYM nodes

The exact specification of ARGS sequence has not yet been decided.

In cases where a pragma must be referred to in the program tree, it is referred to by a "pragma" operator in the tree, as shown in figure Ada-2-2.

```
label:   TREE_NODE
         (OP pragma)
         (SUBNODES PRAGMA_SYM-label:)
```

Figure Ada-2-2: Reference to PRAGMA_SYM node in the tree

Ada-2.8 Reserved words

Ada-3. Declarations and Types

Ada-3.1 Declarations

Ada-3.2 Object declarations

Declarations of variables is discussed in section Ada-4.3.

Ada-3.3 Type and SubType declarations

Ada is a strongly typed language; every variable and expression has a type. Overloaded operators, procedures and functions are disambiguated based on the types of their operands or arguments. The Front End may require a richer representation of type information in order to handle type checking and overloading disambiguation; what is specified here is the representation required as input to the remainder of the compiler.

> Many different relationships may be required in a compiler, particularly for efficiently locating related information for types. Thus, it may be desirable to have all subtypes and derived types refer back to the root type from which they all have come. $TCOL_{Ada}$ specifies the minimum acceptable TCOL for the remainder of the compiler. Information which may be specific to a particular implementation, and which can be regenerated from the $TCOL_{Ada}$ given in this document, is not part of this specification. An implementation which claims to take $TCOL_{Ada}$ as input must accept what this document specifies. However, as shown in figure 4-2, a particular implementation may, internally, accept a richer TCOL.

The remainder of the compiler requires access to the type information for a number of reasons; the representation of type information here is sufficient for these needs. The reaons include range and subscript checking, constraint checking, variant records and discriminants and attribute inquiries.

```
label:  TYPE_SYM
        (KIND DECLARED | SUBTYPE | DERIVED | PREDEFINED)
        (NAME NAME_NODE-label:)
        (CONSTRAINT CONSTRAINT_REP-label-sequence)
        (PARENT TYPE_SYM-label:)
        (REP label:)                      ! ACCESS_REP,
                                          ! ARRAY_REP,
                                          ! ENUMERATION_REP,
                                          ! RECORD_REP,
                                          ! SCALAR_REP
        (PACKING YES | NO)                ! Ada-13.2
        (LENGTH integer)                  ! Ada-13.2


label:  CONSTRAINT_REP
        (RANGE expr-label: expr-label:)
        (ACCURACY expr-label:)
```

Figure Ada-3-1: TYPE_SYM and CONSTRAINT_REP nodes

The ACCURACY attribute is present only on CONSTRAINT_REP nodes for variables whose type is FIXED or FLOAT; for FIXED nodes it is the delta and for FLOAT nodes it is the digits.

Ada-3.4 Derived types

The TYPE_SYM node for a derived type is described in section Ada-3.3 and is identical to the TYPE_SYM node shown there except the KIND attribute is DERIVED. The PARENT attribute refers to the TYPE_SYM node from which this type has been derived.

Ada-3.5 Scalar types

```
label:   SCALAR_REP
         (VARIETY FIXED | FLOAT | INTEGER | CHARACTER | BOOLEAN)
```

Figure Ada-3-2: SCALAR_REP nodes

The number of types in the VARIETY is implementation-dependent, and may also include LONG_REAL, SHORT_INTEGER, etc., but only if these explicit representations are specified in the source text, or as a consequence of a representation decisions made in some separate compilation. Ordinarily, the Front End may only indicate the types suggested by the source text, and the machine-dependent part of the compiler which follows the Front End decides the exact representation suitable for a particular machine.

Ada-3.5.1 Enumeration types

The REP attribute of the TYPE_SYM node for an enumeration type points to an ENUMERATION_REP node.

```
label:   ENUMERATION_REP
         (LITERALS VARBL_SYM-label-sequence)
```

Figure Ada-3-3: ENUMERATION_REP node

The CONSTRAINT_REP node of the TYPE_SYM node specifies the constraint on the enumeration, in terms of the 'ORD attribute, and thus must be in the range from 1 to the size of the enumeration, independent of any special representation given for the type. Thus, the constraints of the root node of an enumeration type E are E'ORD(E'FIRST) and E'ORD(E'LAST). A subtype or derived type of the enumeration type will have its constraints specified in terms of the 'ORD attribute of the root type, as shown in figure Ada-3-4.

```
type COINS is (CENT, NICKEL, DIME, QUARTER, HALF);
        -- TCOL_Ada constraints would be 1..5
type SILVER is new COINS range DIME..HALF;
        -- TCOL_Ada constraints would be 3..5
subtype METER_SILVER is SILVER range DIME..QUARTER;
        -- TCOL_Ada constraints would be 3..4
for COINS use (CENT => 1, NICKEL => 5, DIME => 10,
            QUARTER => 25, HALF => 50);
        -- this declaration would not change the constraints
```

Figure Ada-3-4: Derived types and subtypes of an enumeration type

Ada-3.5.2 Character types

A character type is represented by a TYPE_SYM node which specifies the constraints, and whose REP attribute points to a SCALAR_REP node whose VARIETY is CHARACTER.

Ada-3.5.3 Boolean type

A Boolean type is represented by a TYPE_SYM node which specifies the constraints, and whose REP attribute points to a SCALAR_REP node whose VARIETY is BOOLEAN.

Ada-3.5.4 Integer type

An integer type is represented by a TYPE_SYM node which specifies the constraints, and whose REP attribute points to a SCALAR_REP node whose VARIETY is INTEGER. No commitment to a representation, such as LONG_INTEGER or SHORT_INTEGER is made by the Front End.

Ada-3.5.5 Real types

A real type is represented by a TYPE_SYM node which specifies the constraints and whose REP attribute points to a SCALAR_REP node whose VARIETY is FIXED or

FLOAT. No commitment to a particular representation, e.g., LONG_FLOAT or SHORT_FIXED, is made by the Front End.

A literal whose type is one of the real types is represented by a VARBL_SYM node whose NAME attribute refers to a NAME_NODE whose NAME attribute is the string the user typed in the source program. Thus, "5.0", "5", "5.000" etc. all have separate NAME_NODEs. Once a representation is chosen, many of these literals may be pooled because they will actually have the same representation. However, this is a decision which is bound *after* the Front End processing.

The reason this is done is so the parser and Front End may remain machine-independent, and in particular not be required to do conversions of real types to some particular representation.

The intent is that later phases of the compiler which have knowledge about the target machine representation may generate the internal value by scanning the string in the VARBL_SYM node. To have done the string-to-real (the 'VAL attribute in Ada) and then done a real-to-string (the 'REP attribute in Ada) in the arithmetic supported on the machine on which the parser runs could introduce numeric errors which are unacceptable.

An alternative representation suitable for Ada programs is to represent the value as an expression in terms of the 'VAL attribute, where the operand of 'VAL is the source string representation. See section Ada-4.8; this section explains why a static expression may not require actual evaluation of the operands, which justifies the deferring of evaluation of static expressions involving real literals to a phase after the semantic analyzer.

Ada-3.6 Array types

The REP attribute of a TYPE_SYM node for an array type points to an ARRAY_REP node.

```
label:    ARRAY_REP
          (COMPONENT TYPE_SYM-label:)
```

Figure Ada-3-5: ARRAY_REP nodes

In the TYPE_SYM node for an array type, the CONSTRAINT_REP attribute points to a sequence of TYPE_SYM nodes which specify the constraints on the indices of the array. The REP attribute points to the ARRAY_REP node.

If the array is a subtype or derived type of an array type, the REP attribute is not specified and the PARENT attribute refers to the TYPE_SYM node of which this array is a subtype or derived type.

For a particular implementation, it may be desirable to define the REP attribute for subtypes of the array type to point to the same ARRAY_REP node as the root type; this, however, is an implementation decision for a particular compiler. $TCOL_{Ada}$ requires that the REP attribute of a subtype or derived type of an array be unspecified.

For arrays which are subtypes or derived types of some other array type, a complete CONSTRAINT_REP list must be specified, even if some or all of the constraints on the indices are the same as the parent type.

Since the TCOL representation of an Ada program is a graph, the CONSTRAINT attribute of a subtype may point to the same CONSTRAINT_REP nodes as the parent type when the constraints are identical.

Ada-3.6.1 Index ranges of arrays

Ada-3.6.2 Aggregates

An aggregate is represented by a TREE_NODE whose operator is "aggregate" and whose subnodes are TREE_NODEs whose operator is "agg-choice", as shown in figure Ada-3-6.

```
label:  TREE_NODE
        (OP aggregate)
        (SUBNODES TREE_NODE-label-sequence)

label:  TREE_NODE
        (OP agg-choice)
        (SUBNODES TREE_NODE-label-sequence TREE_NODE-label)
                  ! to TREE_NODEs for simple-expressions
                  ! TYPE_SYM nodes for ranges
                  ! or TREE_NODE whose operator is "others"
```

Figure Ada-3-6: Array aggregate representation in $TCOL_{Ada}$

In the agg-choice operator nodes, the last subnode is the value to be assigned, and the first sequence of subnodes are the indices for which that value is to be assigned. In the case where explicit choices were not present in the source language, an explicit choice must be supplied by the Front End. See figures Ada-3-7 and Ada-3-8.

```
B : TABLE := (5, 4, 8, 1, others => 20);
      -- from Ada Reference Manual p. 3-11

agg:     TREE_NODE
         (OP aggregate)
         (SUBNODES first: second: third: fourth: rest:)


first:   TREE_NODE
         (OP agg-choice)
         (SUBNODES one: five:)                 ! 1 => 5

second:  TREE_NODE
         (OP agg-choice)
         (SUBNODES two: four:)                 ! 2 => 4

third:   TREE_NODE
         (OP agg-choice)
         (SUBNODES three: eight:)              ! 3 => 8

fourth:  TREE_NODE
         (OP agg-choice)
         (SUBNODES four: one:)                 ! 4 => 1

rest:    TREE_NODE
         (OP agg-choice)
         (SUBNODES oth: twenty:)               ! others => 20

oth:     TREE_NODE
         (OP others)
```

Figure Ada-3-7: Example of an aggregate in $TCOL_{Ada}$

C : TABLE := (5, 4, 8, 5..7 => 2, 8 | 10 => 3, others => 1);
-- 5, 4, 8, 1, 2, 2, 2, 3, 1, 3

```
agg2:    TREE_NODE
         (OP aggregate)
         (SUBNODES sn1: sn2: sn3: sn4: sn5: sn6:)


sn1:     TREE_NODE
         (OP agg-choice)
         (SUBNODES one: five:)                  ! 1 => 5

sn2:     TREE_NODE
         (OP agg-choice)
         (SUBNODES two: four:)                  ! 2 => 4

sn3:     TREE_NODE
         (OP agg-choice)
         (SUBNODES three: eight:)               ! 3 => 8

sn4:     TREE_NODE
         (OP agg-choice)
         (SUBNODES five-seven: two:)            ! 5..7 => 2

sn5:     TREE_NODE
         (OP agg-choice)
         (SUBNODES eight: ten: three:)          ! 8 | 10 => 3

sn6:     TREE_NODE
         (OP agg-choice)
         (SUBNODES oth: one:)                   ! others => 1

oth:     TREE_NODE
         (OP others)
five-seven:     TYPE_SYM
         (NAME)                                 ! Anonymous type
         (KIND derived)
         (PARENT TYPE_SYM/label:)               ! of object's
                                                ! index type
         (CONSTRAINT c5-7:)
```

```
c5-7:    CONSTRAINT_REP
         (RANGE five: seven:)                    ! 5..7
```

Figure Ada-3-8: Example of a more complex aggregate in $TCOL_{Ada}$

Ada-3.6.3 Strings

Ada-3.7 Record types

The REP attribute in the TYPE_SYM node for a record type points to a RECORD_REP node.

```
label:   RECORD_REP
         (FIELDS label-sequence)                 ! to VARBL_SYM nodes
                                                 ! or TREE_NODE
                                                 ! (op case) nodes
```

Figure Ada-3-9: RECORD_REP nodes

Ada-3.7.1 Constant Record Components and Discriminants

Ada-3.7.2 Variant parts

The variant components of a record are represented by a tree nearly identical to that produced by the case statement (see seçtion Ada-5.5). However, the last operand of each "when" operator, instead of being a TREE_NODE, is a VARBL_SYM node which represents the component of the variant which is selected by the discriminant. Each of these VARBL_SYM nodes is a component in an anonymous record which holds all of the components of the variant. The null component list is specified by a TREE_NODE whose operator is "null"; this is the same representation as used for the null statement. See Ada-5.a.

A subtype or derived type of a record containing a variant is specified by having a different constraint on the variable which is the discriminant.

Ada-3.7.3 Record Aggregates and Discriminant Constraints

A record aggregate is represented as shown in figure Ada-3-10. A TREE_NODE with operator "record-aggregate" refers to a set of subnodes which have the operator "rec-choice". As in array aggregates (section Ada-3.6.2), the TCOL tree must supply any component names which were omitted in the source because positional notation was used. The first subnodes of the "rec-choice" operator node are the names of the components to be assigned to, and the last subnode is an expression representing the value to be assigned.

```
label:  TREE_NODE
        (OP record-aggregate)
        (SUBNODES expr-label-sequence)

label:  TREE_NODE
        (OP rec-choice)
        (SUBNODES expr-label-sequence expr-label:)
                ! component names, value
```

Figure Ada-3-10: $TCOL_{Ada}$ representation of a record aggregate

---

Ada-3.8 Access types

The REP attribute in the TYPE_SYM node for an access type points to an ACCESS_REP node.

label: ACCESS_REP
(ACCESS-OF TYPE_SYM-*label:*)

Figure Ada-3-11: ACCESS_REP node

Ada-4. Names, Variables and Expressions

Ada-4.1 Names

```
label:  NAME_NODE
        (PNAME string)
        (NAMES label-sequence)         ! TYPE_SYM,
                                       ! VARBL_SYM,
                                       ! EXCEPTION_SYM,
                                       ! LABEL_SYM,
                                       ! PACKAGE_SYM,
                                       ! PRAGMA_SYM,
                                       ! SUBPROGRAM_SYM,
                                       ! TASK_SYM
```

Figure Ada-4-1: NAME_NODE nodes

Several NAME_NODEs may have the same print string, i.e., it is not required that there be one and only one NAME_NODE for each unique character string.

A NAME_NODE exists for literal values also; the "name" is the source string written in the user program. This is particularly important for the representation of real literals if cross-compilation or machine-independent parsing is important; the parser either should not or cannot determine the exact representation of a real literal.

Ada-4.1.1 Index components

```
label:  TREE_NODE
        (OP index)
        (SUBNODES expr-label: expr-label-sequence)
```

Figure Ada-4-2: $TCOL_{Ada}$ for indexed component

The first subnode evaluates to the name of an indexed entity. The remaining subnodes evaluate to the indices. For a simple variable, the first subnode would refer to a VARBL_SYM node; for more complex names, such as an indexed component of a record (an array component of a record), a general TCOL expression would be referred to by the first subnode.

Ada-4.1.2 Selected components

Selected components which are

- An entity declared in the visible part of a module
- An entity declared in an enclosing unit
- A user-defined attribute of a type

have already been identified by the Front End, and references to the selection have already been resolved to point to the correct entities. The purpose of selecting these entities is to provide a syntactic and/or semantic specification of which entity, of a possibly ambiguous set of entities, is desired.

For example, as shown in [2] page 4-2, the selected component "DEVICE.READ" would already *refer to the entry node for the task DEVICE.* The name in the NAME_NODE is "READ".

Thus, "selection" in $TCOL_{Ada}$ refers only to selection of record components.

```
label:  TREE_NODE
        (OP component-select)
        (SUBNODES expr-label: VARBL_SYM-label:)
```

Figure Ada-4-3: $TCOL_{Ada}$ **for selected component**

The first subnode evaluates to the name of a record. The second subnode refers to a VARBL_SYM node which names the field in the record.

Ada-4.1.3 Predefined attributes

A predefined attribute generates a unique operator for each attribute. The complete list of operators for $TCOL_{Ada}$ is given in section Ada-A.

Ada-4.2 Literals

See the discussion of literals in section Ada-3.5, particularly for real literals in section Ada-3.5.5.

---

label: LITERAL_REP
(VALUE *LG-literal*)

Figure Ada-4-4: LITERAL_REP nodes

A LITERAL_REP node is referred to *only* by the INITIALIZE attribute of a VARBL_SYM node (see section Ada-4.3). The VALUE of a LITERAL_REP node holds an LG style literal. The interpretation of this literal depends upon the type of the VARBL_SYM node which refers to it.

The only meaningful LG literals which would appear in the VALUE attribute of a LITERAL_REP node are integers and strings. LG does not support "real" (i.e., fixed point or floating point) literals. As discussed in section Ada-3.5.5, such literals must be represented as the source text characters which specified the literal in the program. At some point in the compiler beyond the Front End, the compiler may determine the correct bit pattern for a real literal and represent it as a LITERAL_REP node whose value is the bit pattern (expressed, for example, as an unsigned octal number).

It may also be necessary to express integer values as strings, if the machine on which the compiler runs cannot express integers with the same range as the target machine.

Note that this does not affect the determination of a value as a static expression, since an expression does not have to be *evaluated* in order to determine if it is static.

Ada-4.3 Variables

Ada-4.3.a Named variables

```
label:  VARBL_SYM
        (NAME NAME_NODE-label:)
        (TYPE TYPE_SYM-label:)
        (CONSTANT NO | UNKNOWN | COMPILE | LINK | EXECUTION)
        (BINDING  IN | OUT | INOUT)          ! see text
        (LOCATION expr-label:)
        (LENGTH expr-label:)
        (ALIGNMENT expr-label)
        (INITIALIZE expr-label:)
```

Figure Ada-4-5: VARBL_SYM nodes

The BINDING attribute is present only for VARBL_SYM nodes which represent formal parameters.

The LOCATION specification applies to either variables or record components, and is present only if an explicit representation or address has been specified (Ada reference chapter 13). For a record, it specifies the bit offset at which the component starts, relative to the start of the record; for variables, it specifies the absolute bit address of the start of the variable.

> The Front End must convert the expression in terms of storage units to an expression in terms of bits. This is a symbolic transformation, since the Front End cannot know how many bits comprise a storage unit.

The LENGTH and ALIGNMENT specifications apply only to VARBL_SYM nodes representing record components, and are expressed as bit lengths and bit alignments. See section 13.4 in the Ada Reference Manual.

A literal in the source language is always represented by a VARBL_SYM node whose NAME attribute refers to a NAME_NODE which contains the source language string and whose CONSTANT attribute is COMPILE. The INITIALIZE attribute refers to a LITERAL_REP node which holds the value of the literal.

Ada-4.3.b Slices

A slice is represented in TCOL as shown in figure Ada-4-6. The first subnode refers to an expression which evaluates to the name of an array, subarray, or access object whose value designates an array. The range is represented by the second subnode, which refers to an anonymous TYPE node which is a derived type of the index type of the array, and whose constraints specify the slice.

---

label: TREE_NODE
(OP slice)
(SUBNODES expr-*label:* TYPE_SYM-*label:*) .

Figure Ada-4-6: $TCOL_{Ada}$ representation for an array slice access

---

Ada-4.4 Expressions

Ada-4.5 Operators and Expression Evaluation

---

label: TREE_NODE
(OP *identifier*)
(DEFN *label:*)
(SUBNODES expr-*label-sequence*)

Figure Ada-4-7: TREE_NODE in $TCOL_{Ada}$

The OP attribute contains an LG identifier which indicates the operation.

The DEFN attribute points to a TYPE_SYM node for predefined types, or arrays or records, or points to a SUBPROGRAM_SYM node for the function which implements the operator. This attribute applies only to unary or binary operators as defined in Ada-4, and assignment of predefined types, arrays or records.

The DEFN attribute for predefined scalar types points to a TYPE_SYM node, whose REP field points to a SCALAR node. The information may be extracted by walking this chain of pointers and stored in some implementation-specific field in the TREE_NODE. However, this extension is not required by $TCOL_{Ada}$.

This attribute also permits a user to define a type-specific assignment operator if it were permissible in the source language.

TCOL can have two representations for a unary or binary operator: it can represent them as either function calls of 1 or 2 arguments or as operator nodes in the tree for each operator. In the particular case of predefined types and types which are subtypes or derived types of the predefined types, it is desirable to represent the unary and binary operators as operator nodes in the tree, for purposes of various optimization techniques, e.g., expression reordering, applying associativity, commutativity, or unary complement optimizations, etc.

It is unclear from the semantics of Ada if an overloaded operator such as "+" is expected to preserve these properties, i.e., is "+" associative, commutative, etc; do axioms such as "(A-B) => -(B-A)" hold? It is also unclear whether or not this is also true of user-defined types which are not defined in terms of the predefined types, e.g., arrays, records, etc.

The DEFN attribute allows us to represent operators, even those defined by explicit overloading, as unary or binary tree operators, which greatly simplifies the task of optimization. To actually generate the code for such operators, the DEFN attribute makes the operator definition available.

A code generator may look at the DEFN attribute, or may require that any unary or binary operator defined by a user-declared procedure be transformed into a procedure call node before code generation begins. Such a decision is an implementation strategy in the Back End of the compiler and is made for a particular implementation. Such a transformation is essentially a simple tree transformation.

In this specification of $TCOL_{Ada}$, if the operator token for an operation as defined in this section appears in the tree, e.g., "and", "or", "+", "*", "<", etc., then its conventional arithmetic properties of associativity, distributivity, commutativity, etc. are assumed to be preserved. In the case where semantic analysis wishes to prohibit optimizations which rely on these properties, it must represent the operations as function calls.

In addition to all of the standard operators described in sections Ada-4.5.1 through Ada-4.5.6, there is a special operator, "paren", which is used to indicate associativity across parenthesized expressions is not valid. In any case where the semantic analyzer wishes to block the use of associativity axioms by an optimizing compiler, it can insert this operator in the tree. This allows other properties of the operator node, such as commutativity, to be retained. If the associativity could only be prevented by using the procedure-call representation, other, permissible, optimizations might be also prohibited.

```
label:  LEAF_NODE
        (OP leaf)
        (SUBNODES VARBL-label:)
```

Figure Ada-4-8: LEAF_NODE in $TCOL_{Ada}$

A LEAF_NODE is a particular extension to a TREE_NODE, and is present because in most implementations, the phases in the compiler which follow the Front End wish to place different kinds of information in a LEAF_NODE than in a TREE_NODE. Two attributes which are common to both LEAF_NODEs and TREE_NODEs are the OP and SUBNODES attributes; the OP attribute for a LEAF_NODE always has the operator "leaf".

Ada-4.5.1 Logical Operators

Source	$TCOL_{Ada}$
and	and
or	or
xor	xor

Figure Ada-4-9: Logical operators: source-to-$TCOL_{Ada}$ transformation

In addition, there are two other boolean operators, cand and cor, representing respectively and-then and or-else, which are described in section Ada-5.4.1.

These are currently restricted to the conditional part of an If statement, for no discernable reason. In TCOL, they are valid binary operators on boolean operands.

Ada-4.5.2 Relational and membership operators

Source	$TCOL_{Ada}$
<	<
>	>
<=	<=
>=	>=
=	=
/=	/=
in	in
not in	not-in

Figure Ada-4-10: Relational and membership operators: source-to-$TCOL_{Ada}$

Ada-4.5.3 Adding operators

Source	$TCOL_{Ada}$
+	+
-	-
&	&

Figure Ada-4-11: Adding operators: Source-to-$TCOL_{Ada}$ transformation

Ada-4.5.4 Unary operators

Source	$TCOL_{Ada}$
-	U-
+	U+
not	not

Figure Ada-4-12: Unary operators: source-to-$TCOL_{Ada}$ transformation

Unary plus is represented in $TCOL_{Ada}$ by a unique operator, "U+". The token "+"

as a TCOL operator is permitted to represent *only* the binary addition operator.

Since the identity operator conveys no information, it may be omitted entirely by the semantics phase and not appear in $TCOL_{Ada}$.

Unary minus is represented in $TCOL_{Ada}$ by a unique operator, "U-". The TCOL token "-" is permitted to represent *only* the binary subtraction operator.

The not operator is defined for boolean scalar operands and boolean-array operands; the DEFN attribute will describe which one this represents.

Ada-4.5.5 Multiplying operators

Source	$TCOL_{Ada}$
*	*
/	/
mod	mod

Figure Ada-4-13: Multiplying operators: source-to-$TCOL_{Ada}$ transformation

Ada-4.5.6 Exponentiation operator

Source	$TCOL_{Ada}$
**	**

Figure Ada-4-14: Exponentiation operator: source-to-$TCOL_{Ada}$ transformation

Ada-4.6 Qualified expressions

Qualified expressions serve several purposes. Some of those purposes are purely an interaction at the semantic level, e.g., to disambiguate potentially ambiguous expressions or literals.

In those cases where a qualification carries no semantic information, the qualification may be dropped by the semantic analyzer. An example of such a situation is shown in figure Ada-4-15.

```
type color is (UV VIOLET BLUE GREEN YELLOW ORANGE RED IR BLACK);
type STOPLIGHT is (RED YELLOW GREEN);
 -- without qualification, the following is ambiguous
PRINT(STOPLIGHT(RED));
```

Figure Ada-4-15: Use of a qualified expression

Since, at the output of the semantic analyzer, the literal RED would be uniquely identified, the qualification on the expression would be redundant, and could be eliminated.

However, in figure Ada-4-16, the qualification is important, and must not be removed by the Front End. Since no representation decision has been bound by the Front End (excluding explicit user specifications or specifications forced by separately compiled program units), a conversion from the representation of the subtype to the type of the parent type may be necessary.

```
type X is new integer range 1..65535;
subtype Y is X range 1..7;

A, B : X;
C, D : Y;
      -- statements
      A := X(C) + X(D);
```

Figure Ada-4-16: Qualified expression which may imply run-time type conversion

Ada-4.6.1 Explicit type or Subtype specification

See section Ada-4.6.

Ada-4.6.2 Type conversion

There is no implicit type coercion in $TCOL_{Ada}$; any type conversions must be explicitly represented in the TCOL tree.

Ada-4.7 Allocators

Ada-4.8 Static expressions

A static expression is represented by a VARBL_SYM node (section Ada-4.3) whose CONSTANT attribute is COMPILE and whose INITIALIZE attribute refers to a LITERAL_REP node or an expression whose operands are static expressions.

> At various places, Ada requires static expressions to specify certain values. The semantic analyzer may choose to evaluate expressions ("constant folding") to determine if they are static expressions; however, it need not evaluate any expressions, even though they may be static expressions, if static expressions are not required by the language (e.g., the range constraints on a type or subtype).
>
> In addition, the semantic analyzer may determine if an expression is a *static* expression without actually performing any evaluation, simply by determining, by a recursive tree walk, that all the operands of the expression are themselves static expressions. Ultimately, such a tree walk must reach every LEAF_NODE, which to satisfy the requirement of being a static expression must point to a VARBL_SYM whose CONSTANT attribute is COMPILE and whose INITIALIZE attribute points to a LITERAL_REP node.

Ada-5. Statements

Ada-5.a Null statement

The null statement is represented by a TREE_NODE whose operator is "null", as shown in figure Ada-5-1.

```
null:   TREE_NODE
        (OP null)
```

Figure Ada-5-1: null statement

Ada-5.b Statement sequences

A sequence of statements is represented by an *n*-ary tree node whose operator is ";" and whose subnodes are each of the statements in the sequence. If a ";" node happens to have only a single subnode, a reference to the ";" node may be replaced by a reference to the subnode. This transformation is permitted to any phase of the compiler beyond the parser, including the semantics phase.

An example of the two alternate representations of a sequence are shown in figure Ada-5-2; in this example, the operator is some n-ary operator which can refer to a statement sequence.

```
top:    TREE_NODE
        (OP identifier)
        (SUBNODES label: label: e:)

e:      TREE_NODE
        (OP ;)
        (SUBNODES s3:)

s3:     ! not shown for this example
```

or, alternatively

```
top:    TREE_NODE
        (OP identifier)
        (SUBNODES label: label: s3:)
```

Figure Ada-5-2: Permissible representations for statement sequences

"Flattening" of such tree nodes is permissible; that is, if any subnode of a ";" operator tree node refers to another ";" node, the reference may be replaced with the subnodes of the node referred to, as shown in figure Ada-5-3.

```
stmnt:  TREE_NODE
        (OP ;)
        (SUBNODES ... s0: ...)


s0:     TREE_NODE
        (OP ;)
        (SUBNODES s1: s2: s3:)
```

may be replaced by:

```
stmnt:  TREE_NODE
        (OP ;)
        (SUBNODES ... s1: s2: s3: ...)
```

Figure Ada-5-3: Flattening of ";" operator nodes

Ada-5.c Statement Labels

The label of a statement may be used either as the destination of a goto statement, or if the statement is a loop statement, as the operand of an exit statement. A label is represented in TCOL as a LABEL_SYM node; because the use of a label in a goto and exit are different, an Ada label may generate two label nodes, one for the "goto" label and one for the "exit" label. In addition, the program tree contains two operators, "gotolabel" and "exitlabel", which mark the point in the program tree where the label appears. Their form is shown in figure Ada-5-4.

These TREE_NODEs are used by the code generator, to determine when to emit the label in the code stream. In addition, compilers which do flow analysis require these nodes so that the program graph may be constructed.

```
label: TREE_NODE
        (OP gotolabel)
        (SUBNODES LABEL_SYM-label: expr-label:)

label: TREE_NODE
        (OP exitlabel)
        (SUBNODES LABEL_SYM-label: expr-label:)
```

Figure Ada-5-4: $TCOL_{Ada}$ tree for gotolabel and exitlabel operators

---

A simple "gotolabel" is shown in figure Ada-5-6, while a label which is both a "gotolabel" and an "exitlabel" is shown in figure Ada-5-7.

```
label:    LABEL_SYM
          (NAME NAME_NODE-label:)
          (TREE expr-label:)                 ! (OP gotolabel) or
                                             ! (OP exitlabel)
```

Figure Ada-5-5: LABEL_SYM nodes

```
-- statements
if A < B then goto Z end if;
-- statements
<<Z>> A := XYZ;
-- statements

zlb:     LABEL_SYM
         (NAME zname:)
         (TREE agets:)


zname:   NAME_NODE
         (PNAME "Z")
         (NAMES ... zlb: ...)

pgm:     TREE_NODE
         (OP ;)
         (SUBNODES ... test: ... agets: ...)

test:    TREE_NODE
         (OP if)
         (SUBNODES  cond: go:)

cond:    ! not shown, boolean condition

go:      TREE_NODE
         (OP goto)
         (SUBNODES zlb:)

agets:   TREE_NODE
         (OP gotolabel)
         (SUBNODES assgn:)

assgn:   TREE_NODE
         (OP :=)
         (SUBNODES ...)
```

Figure Ada-5-6: LABEL_SYM and goto

```
goto L;
-- other statements
<<L>> loop
    -- statements
    exit L when e0;
    -- statements
   end loop L;
```

```
lname:  NAME_NODE
        (NAME "L")
        (NAMES ... elb: glb: ...)


elb:    LABEL_SYM
        (NAME lname:)
        (TREE elab:)

glb:    LABEL_SYM
        (NAME lname:)
        (TREE glab:)

pgm:    TREE_NODE
        (OP ;)
        (SUBNODES ... go: ... glab: ...)

go:     TREE_NODE
        (OP goto)
        (SUBNODES glb:)

glab:   TREE_NODE
        (OP gotolabel)
        (SUBNODES glb: elab:)

elab:   TREE_NODE
        (OP exitlabel)
        (SUBNODES elb: body:)

body:   TREE_NODE
        (OP loop)
        (SUBNODES ... exit: ...)
```

```
exit:   TREE_NODE
        (OP exit)
        (SUBNODES e0: elb:)
```

Figure Ada-5-7: Interactions with LABEL_SYM nodes

Ada-5.1 Assignment statements

```
label:  TREE_NODE
        (OP :=)
        (SUBNODES expr-label: expr-label:)
              ; to destination, expression trees
```

Figure Ada-5-8: $TCOL_{Ada}$ tree for assignment

The first subnode of the assignment operator evaluates to the location to perform the assignment. This may be an aribtrarily complex expression which could include array subscripting and component selection.

The semantics of an assignment in Ada is that it is always checked. The pragma to suppress the RANGE_ERROR exception will appear in the DECLARATION_INFO node of a block, subprogram, task, etc., and is taken as advice to the compiler to suppress the exception. Whether or not the compiler chooses to honor this pragma is an implementation decisions which is not in the domain of the Front End; therefore, the Front End does not include any explicit checking of the assignment nor does it suppress any implicit checking of the assignment.

Ada-5.1.1 Array and Slice assignment

See section Ada-4.3.b.

Ada-5.1.2 Record assignments

Ada-5.2 Subprogram calls

```
label:    TREE_NODE
          (OP call)
          (SUBNODES SUBPROGRAM_SYM-label: expr-label-sequence)
```

Figure Ada-5-9: Subprogram call operator

Ada-5.2.1 Actual parameter associations

TCOL$_{Ada}$ requires that each call provide the correct number of actual parameters in the correct positional order. Thus, the use of "keyword" parameters, where the parameter names are supplied explicitly, is resolved during semantic analysis, and the actual call TREE_NODE contains the parameters in the same order as the formal parameters of the procedure declaration. See also section Ada-5.2.2.

Ada-5.2.2 Omission of actual parameters

When an actual parameter may be omitted because the subprogram declaration provides a default value, a mechanism must exist so the procedure call can provide the correct value. As described in section Ada-5.2.1, the call must provide all of the actual parameters in the correct order. Furhtermore, the value of the default is determined by elaborating the expression at the time the procedure declaration is elaborated, so the value must be stored so subsequent procedure calls can use it.

> As an optimization, the later phases of the compiler may determine that no call of the procedure omits the parameter, so the default need not be evaluated since it is never used. However, this decision cannot usually be made by the Front End. Because of interactions with separate compilation, it may not be possible to determine if this optimization is possible except in some very restricted cases.

When an actual parameter may be omitted because the subprogram declaration has specified a default value, the DECLARATION_INFO node for the block which contains the subprogram includes a dummy VARBL_SYM node which identifies a runtime location to hold the value of the default parameter expression. The default parameter expression is elaborated when the declarations are processed, and the result of the elaboration is stored in the location named by this dummy VARBL_SYM node. A call of the subprogram for which the actual parameter corresponding to this VARBL_SYM node has been omitted will contain, for the parameter expression, an expression which refers to the VARBL_SYM node.

```
declare
  -- other declarations
  procedure DEF1(parm : in color := My_Favorite_Color) is
                -- procedure body
                ;
     -- My_Favorite_Color is not a static expression
     -- and is a variable visible at this level
  -- more declarations
  begin
    -- program text
    DEF1;
    -- program text
  end;

decls:   DECLARATION_INFO
         (SUBPROGRAMS def1:)
         (VARBLS ... dummy: ...)

def1:    SUBPROGRAM_SYM
         (PARAMETERS ... parm: ...)

parm:    VARBL_SYM
         (INITIALIZE dummy:)

dummy:   LEAF_NODE
         (OP leaf)
         (SUBNODES d-v:)

d-v:     VARBL_SYM
         (INITIALIZE fav-exp:)

fav-exp: LEAF_NODE
         (OP leaf)
         (SUBNODES my-fav:)

my-fav:  VARBL_SYM
         (NAME ...)        ! "My_Favorite_Color"
         ! ... etc.
```

```
callit: TREE_NODE
        (OP call)
        (SUBNODES def1: dummy:)
```

Figure Ada-5-10: Default parameter representation

Ada-5.2.3 Restrictions on subprogram calls

The Front End has the responsibility for checking the TYPE_SYM consistency between procedure actual parameters and procedure formal parameters. The constraints, if they are represented by static expressions, may be checked by the Front End, but this is not required. The checking of constraints at the time of the call is implicit, in the same way the checking of constraints during assignment is implicit; a code generator may or may not honor the RANGE_ERROR pragma.

A compiler may determine that the raising of an exception is either always the case or never the case at subprogram call time, and as for assignment, may choose to eliminate the code to test for the exception and either always raise it or never raise it, as appropriate. However, this optimization should not be made by the Front End.

Ada-5.3 RETURN statement

return

```
label:  TREE_NODE
        (OP return)
        (SUBNODES SUBPROGRAM_SYM-label:)
```

Figure Ada-5-11: $TCOL_{Ada}$ tree for return statement

```
        return e0;       -- expression e0

label:  TREE_NODE
        (OP return-value)
        (SUBNODES SUBPROGRAM_SYM-label: expr-label:)
```

Figure Ada-5-12: $TCOL_{Ada}$ tree for return statement for value return

Restrictions on **return** statements are assumed to be enforced by the Front End, in the sense that a return operator node will always generate code to return from the subprogram, even if, for some reason, it appeared in a context in which the language forbids this. If the procedure returns a value, the **return** statement is checked by the Front End for conformity to the type restrictions of the return value; a return-value operator that returns a result, or a return operator which *does not*, are both assumed by the Back End to be valid in their context. The phases of the compiler beyond the Front End assume that necessary checking has been done by the syntax and semantic analyzers.

Ada-5.4 If statements

```
label:  TREE_NODE
        (OP if)
        (SUBNODES expr-label: expr-label: expr-label:)
```

Figure Ada-5-13: $TCOL_{Ada}$ tree for if statement

The **if** statement produces a ternary node whose first subnode is the condition, whose second is the **then** clause and whose third is the **else** clause.

The Front End treats the elsif clauses as else clauses, and transforms the **if** statement to a sequence of nested **if** statements. Any if operator nodes generated from the elsif clauses have the operator "elsif".

In general, the processing of an "elsif" operator and an "if" operator in the back end of the compiler will be identical; the distinction is made for those cases in which the additional knowledge might be used to some advantage.

```
if e0
      then s1
   elsif e2
      then s3
   elsif e4
      then s5
   else s6
end if;

if:      TREE_NODE  (OP if)
         (SUBNODES e0: s1: elf1:)

elf1:    TREE_NODE  (OP elsif)
         (SUBNODES e2: s3: elf2:)

elf2:    TREE_NODE  (OP elsif)
         (SUBNODES e4: s5: s6:)
```

Figure Ada-5-14: $TCOL_{Ada}$ tree for elsif clauses

If no else clause is present, a dummy TCOL node for a null statement must be supplied by the Front End, so that every TREE_NODE with an "if" or "elsif" operator has three subnodes; the boolean expression, the statements from the then clause and the statements from the else clause.

Ada-5.4.1 Short-circuit conditions

The condition of an if is one of the forms:

expression

expression and then *expression*

expression or else *expression*

the short circuit operators shown in figure Ada-5-15 are represented as shown in figure Ada-5-16.

Source	$TCOL_{Ada}$
and then	cand
or else	cor

Figure Ada-5-15: Short-circuit boolean operators: Source-to-$TCOL_{Ada}$

```
label:  TREE_NODE
        (OP cand)                               ! and then
        (SUBNODES expr-label: expr-label:)

label:  TREE_NODE
        (OP cor)                                ! or else
        (SUBNODES expr-label: expr-label:)
```

Figure Ada-5-16: $TCOL_{Ada}$ representation of short-circuit boolean operators

There seems to be no good reason for the restriction of these operators to boolean conditions in if statements. This representation demonstrates that they can easily be handled as binary operators.

Ada-5.5 Case statement

```
case e0 of
   when e1..e2 => s3
   when e4 | e5 => s6
   when others => s7;


label:   TREE_NODE  (OP case)
         (SUBNODES expr-label: expr-label-sequence)
                  ! to TREE_NODE of case index expression, and
                  ! to TREE_NODEs for each case


label::  TREE_NODE  (OP when)
         (SUBNODES expr-label-sequence expr-label:)
                  ! sequence refers to TREE_NODEs or
                  ! TYPE_SYM nodes or TREE_NODE with "others"
                  ! operator

! choice: others
label:   TREE_NODE  (OP others)
```

Figure Ada-5-17: TCOL$_{Ada}$ **tree for case statement**

The semantics of a "when" operator node are that the last expression is the one to execute if any of the preceding choice expressions matched e0.

A choice may be represented by one of the following:

- A TREE_NODE which produces a single value.

- A TYPE_SYM node which represents a range; an anonymous TYPE_SYM node will be created to represent each range, and will be a derived type of the type of the case index.

- A TREE_NODE whose operator is "others". This TREE_NODE has no subnodes.

Type-checking between the case index e0 and the selectors in the when clauses is the responsibility of the semantics phase. Optimizations, for example, constant folding to eliminate unreachable cases, may be done by the semantics phase, but this is not required.

Ada-5.6 Loop statements

Ada-5.6.a loop statement

---

loop
    -- body
end loop;

```
label:    TREE_NODE
          (OP loop)
          (SUBNODES expr-label:)
```

Figure Ada-5-18: $TCOL_{Ada}$ **tree for loop statement**

The LOOP operator implies a "loop forever" which may be terminated only by some explicit control transfer, e.g., exit, goto. The subnode of a loop operator tree node is the body of the loop.

Ada-5.6.b while statement

---

while
    -- condition
    loop
        -- body
    end loop;

```
label:    TREE_NODE
          (OP while)
          (SUBNODES expr-label: expr-label:)
```

Figure Ada-5-19: $TCOL_{Ada}$ **tree for while statement**

The statements in the body are performed while the condition is true.

Ada-5.6.c for statement

```
for var in [reverse] discrete_range
   loop
      -- body
   end loop

label:   TREE_NODE
         (OP for-up | for-down)
         (SUBNODES VARBL_SYM-label: TYPE_SYM-label: expr-label:)
```

Figure Ada-5-20: $TCOL_{Ada}$ tree for for statement

The first subnode points to a VARBL_SYM node which holds the value for each iteration. The second subnode points to a TYPE_SYM node which specifies the range of the iteration. In the case where other than a type-mark is given to specify the range, an "anonymous type" is created to represent the range, and the subnode refers to the TYPE_SYM node for this anonymous type. The third subnode refers to the program tree representing the body of the loop.

Ada-5.7 exit statements

```
label:   TREE_NODE
         (OP exit)
         (SUBNODES expr-label: LABEL_SYM-label:)
```

Figure Ada-5-21: $TCOL_{Ada}$ tree for exit statment

If the condition is omitted in the source program, the Front End provides a reference to a constant expression whose value is "true".

If the exit applies to an unlabelled construct, the Front End must supply a dummy LABEL_SYM *node* and a TREE_NODE whose operator is "exitlabel" in the appropriate place in the tree. See section Ada-5.c.

Ada-5.8 goto statement

```
label:   TREE_NODE
         (OP goto)
         (SUBNODES LABEL_SYM-label:)
```

Figure Ada-5-22: $TCOL_{Ada}$ tree for goto statement

If the program label created two LABEL_SYM nodes, the target LABEL_SYM node for a goto statement is the LABEL_SYM node whose TREE_NODE refers to a tree node whose operator is "gotolabel". See Ada-5.c.

Ada-5.9 Assert statement

```
label:   TREE_NODE
         (OP assert)
         (SUBNODES expr-label:)
                 ! to TREE_NODE for condition
```

Figure Ada-5-23: $TCOL_{Ada}$ tree for assert statement

Ada-6. Declarative parts, subprograms and blocks

Ada-6.1 Declarative parts

```
label:  DECLARATION_INFO
        (SUBPROGRAMS SUBPROGRAM_SYM-label-sequence)
        (VARBLS VARBL_SYM-label-sequence)
        (TYPES TYPE_SYM-label-sequence)
        (EXCEPTIONS EXCEPTION_SYM-label-sequence)
        (PRAGMAS PRAGMA_SYM-label-sequence)
        (TASKS TASK_SYM-label-sequence)
        (PACKAGES PACKAGE_SYM-label-sequence)
        (ELABORATION_ORDER label-sequence)        ! to all nodes in
                                                  ! above attributes
```

Figure Ada-6-1: DECLARATION_INFO node in $TCOL_{Ada}$

The DECLARATION_INFO node specifies all of the declarations to be elaborated in the declaration list. The attributes SUBPROGRAM_SYMS, VARBLS, etc. are used to group declarations of one kind. However, since order is important (a VARBL_SYM node may be used in the later elaboration of a PROCEDURE or TYPE_SYM node, for example), the ORDER attribute points to each object to be elaborated in the order in which they must be elaborated.

† TYPE_SYM nodes must be elaborated in order to evaluate the bounds constraints. PROCEDURE nodes must be elaborated to ascertain the value of default parameters, since the value of a default parameter is determined at the time the procedure declaration is elaborated, not at procedure call time.

While it is true that some declarations need not be elaborated, (for example, declarations involving static expressions), elimination of such nodes from the DECLARATION_INFO node or the ORDER list is strictly an issue of attempting to optimize compiler performance. Such an optimization is solely related to a particular implementation of a compiler, and is not to be performed by the Front End.

Ada-6.2 Subprogram declaration

```
label:  SUBPROGRAM_SYM
        (NAME NAME_NODE-label:)
        (BODY expr-label-sequence)
        (RESULT TYPE_SYM-label:)
        (KIND  PROCEDURE | VALUE-PROCEDURE | FUNCTION
                         | ENTRY | TASK-BODY)
        (PARAMETERS VARBL_SYM-label-sequence)
        (LINKAGE LINKAGE-label:)
        (PRAGMAS PRAGMA_SYM-label-sequence)
        (DECLARATIONS DECLARATION_INFO-label:)
        (EXCEPTION expr-label:)
        (LOCATION expr-label:)
```

Figure Ada-6-2: SUBPROGRAM_SYM node in $TCOL_{Ada}$

The BODY attribute refers to only a single body for all subprograms except ENTRY subprograms; for ENTRY subprograms, a sequence of zero or more body labels may be given. See Ada-9.5.

LINKAGE nodes are not yet specified. They contain information about the type of linkage to be used to call the procedure; this captures the information required to interface to various languages. In addition, particular Ada implementations may use different calling conventions for procedures forming the run-time system primitives.

The RESULT attribute is present only for functions and value-returning procedures, and indicates the type of the result which they return.

The LOCATION attribute is present only for subprograms for which an explicit address specification has been supplied; see section Ada-13.5.

Ada-6.3 Formal parameters

Formal parameters are represented by VARBL_SYM nodes which are referred to by the PARAMETERS attribute of a SUBPROGRAM_SYM node (Ada-6.2). The VARBL_SYM nodes also specify the binding of the parameters; see Ada-4.3. The INITIALIZE attribute of an in PARAMETER which has a default value is specified by having the INITIALIZE attribute point to an expression which is used to determine the value to be passed by a call on the subprogram. This expression, because of the semantics of Ada, will always refer to a dummy variable created at procedure declaration elaboration time and which holds the value computed at that time. The dummy VARBL_SYM node is the the VARBLS list of the DECLARATION_INFO node associated with the subprogram, and *its* INITIALIZE attribute refers to the expression to be elaborated at procedure declaration time. For an example of all of this, see section Ada-5.2.2.

Ada-6.4 Subprogram bodies

```
label:  TREE_NODE
        (OP procedure)
        (SUBNODES SUBPROGRAM_SYM-label: expr-label:)

label:  TREE_NODE
        (OP value-procedure)
        (SUBNODES SUBPROGRAM_SYM-label: expr-label:)

label:  TREE_NODE
        (OP function)
        (SUBNODES SUBPROGRAM_SYM-label: expr-label:)

label:  TREE_NODE
        (OP task)
        (SUBNODES SUBPROGRAM_SYM-label: expr-label:)

label:  TREE_NODE
        (OP package)
        (SUBNODES SUBPROGRAM_SYM-label: expr-label:)
```

Figure Ada-6-3: TREE_NODEs for subprogram bodies

The first subnode of a subprogram body is the SUBPROGRAM_SYM node. The second subnode of a subprogram body is a pointer to the tree which represents the code of the body.

The specification of accept statements and their bodies is in section Ada-9.5.

The reason for the existence of such nodes in the program tree representation is to simplify the code generator; when such a node is encountered by the code generator, the prolog and epilog code will be emitted (as appropriate in the treewalk).

Ada-6.5 Function subprograms

See Ada-6.4.

Ada-6.6 Overloading of subprograms

Ada-6.6.1 Overloading of operators

$\mathrm{TCOL}_{\mathrm{Ada}}$ requires that the semantics phase perform disambiguation on overloaded operators. Thus, every operator in the tree is uniquely identified with the particular implementation of that operator. If the operator is a user-defined operator, it may be represented either as a subprogram (function) call or as a binary or unary operator as given in section Ada-4. If it is represented as an operator node, the DEFN attribute of the TREE_NODE for that operator points to the definition of the function. This representation permits the standard arithmetic interpretations to be placed on all the operators, e.g., "+" is associative and commutative, and obeys the distributive law with respect to "*"; "<" is a total ordering relationship whose complement is ">=", etc.

Because the DEFN attribute points to code which implements the operator, or to some other definition (such as for builtin operators on integer types), the same token, "+", can be used to represent many types of addition for which the standard interpretations hold.

At some later stage in the compilation process, the TCOL operator may be uniquely identified such that real arithmetic, integer arithmetic, etc. all have unique TCOL operators in that dialect of TCOL. When, or if, this sort of transformation is done depends upon the particular compiler implementation.

Ada-6.7 Blocks

A block is represented by a TREE_NODE whose operator is "block" and whose subnodes refer to the DECLARATION_INFO node for the block and the tree which describes the body of the block.

```
label:  TREE_NODE
        (OP block)
        (SUBNODES DECLARATION_INFO-label: expr-label:)
```

Figure Ada-6-4: $TCOL_{Ada}$ for a block

Ada-7. Modules

The specification of modules specifies effects at syntax analysis and semantic analysis time. The results of semantic analysis, and in particular, visibility of variables or their representations (private declarations) are all implicit in the $TCOL_{Ada}$ tree.

Ada-7.1 Module structure

Ada-7.2 Module specifications

Ada-7.3 Module bodies

See Ada-6.4.

Ada-7.4 Private type declarations

Ada-8. Visibility rules

The scope and visibility of a name are determined by the semantic analyzer. All cases of overlapping scope are resolved, and the TCOL representation always refers to the correct identifier; there is no concept of overlapping scope or overloaded identifiers in the TCOL representation.

Ada-8.1 Scope of Declarations

Ada-8.2 Visibility of Identifiers

Ada-8.3 Restricted Program Units

Ada-8.4 USE clauses

Ada-8.5 Renaming

The effects of renaming on the TCOL representation have not yet been specified.

Ada-8.6 Predefined Environment

Ada-9. Tasks

Ada-9.1 Task declarations and task bodies

```
label:    TASK_SYM
          (NAME NAME_NODE-label:)
          (DECLARATION DECLARATION_INFO-label:)
          (BODY SUBPROGRAM_SYM-label:)
```

Figure Ada-9-1: TASK_SYM node

The TASK_SYM node refers to a DECLARATION_INFO node which contains the declarations for the task. The BODY attribute refers to a SUBPROGRAM_SYM node for the task body; see section Ada-6.2.

This specification is preliminary.

Ada-9.2 Task hierarchy

Ada-9.3 Task initiation

```
label:    TREE_NODE
          (OP initiate)
          (SUBNODES expr-label-sequence)
```

Figure Ada-9-2: TCOL$_{Ada}$ representation for initiate

The subnodes of an initiate operator node are either task designators which are a single task name (LEAF_NODEs pointing to TASK_SYM nodes) or task designators which specify one or more members of a family of tasks. For a single member, the form is as shown in figure Ada-9-3, and for several members, the form is as shown in figure Ada-9-4. If the source language specifies the name of a task family, the

TCOL tree represents the explicit range which for a family of tasks T runs from T'FIRST to T'LAST.

```
task T(1..10) is
      -- task declarations
end T;
task body T is
      -- body
end T;
initiate T(4), T(6);
```

```
ini:     TREE_NODE
         (OP initiate)
         (SUBNODES t4: t6:)

t4:      TREE_NODE
         (OP index)
         (SUBNODES task-T four:)

t6:      TREE_NODE
         (OP index)
         (SUBNODES task-T six:)

task-T:  TASK_SYM
         ! ... etc.
```

Figure Ada-9-3: $TCOL_{Ada}$ tree for initiating single members of a task family

---

```
      initiate T;      -- T as in figure Ada-9-3

ini:     TREE_NODE
         (OP initiate)
         (SUBNODES all-t:)
```

```
all-t:  TREE_NODE
        (OP slice)
        (SUBNODES task-T: one-ten:)            ! T(1..10) explicit!

one-ten: TYPE_SYM
        ! a derived type of the index range of the task
        ! family, with the constraints 1..10
```

Figure Ada-9-4: $TCOL_{Ada}$ tree for initiating a family of tasks

---

Ada-9.4 Normal termination of tasks

Ada-9.5 Entry declarations and Accept statements

An entry declaration generates a SUBPROGRAM_SYM node (section Ada-6.2) whose KIND is ENTRY and which contains multiple BODY pointers; there is one pointer to each body of an accept statement.

```
label:  TREE_NODE
        (OP accept)
        (SUBNODES SUBPROGRAM_SYM-label: expr-label:)
```

Figure Ada-9-5: $TCOL_{Ada}$ form of accept statement

Ada-9.6 DELAY statement

```
label:  TREE_NODE
        (OP delay)
        (SUBNODES expr-label:)
```

Figure Ada-9-6: $TCOL_{Ada}$ representation for the delay statment

---

Ada-9.7 SELECT statement

The exact representation for the select statement is not yet specified.

Ada-9.8 Task priorities

Ada-9.9 Task and Entry attributes

Ada-9.10 abort statements

```
label:   TREE_NODE
         (OP abort)
         (SUBNODES expr-label-sequence)
                  ! to same types of nodes as an
                  ! initiate statement
```

Figure Ada-9-7: $TCOL_{Ada}$ representation for **abort** statement

Ada-9.11 Signals and Semaphores

Ada-10. Program structure and compilation issues

$TCOL_{Ada}$ does not normally specify the representation of data items except when this is explicit in the source code. However, knowledge from previous separate compilations, in which representation decisions have been bound, has the same effect as an explicit representation specification, in that the remaining phases of the compiler are not permitted to select a new representation.

It is therefore necessary for the information about representation choices be made available to the Front End when separate compilation is done, so that the Front End may bind any representation decisions which may not be changed. This requires a specification of what information is required for separate compilation, and a specification of how to generate this information from some later form of the TCOL tree. Such a specification is beyond the scope of this document.

Ada-10.1 Compilation units

Ada-10.2 Subunits of compilation units

Ada-10.3 Order of compilation

Ada-10.4 Program library

Ada-10.5 Elaboration of compilation units

Ada-10.6 Program optimization

Although static expressions *may* be evaluated by the compiler Front End, there is no *requirement* that this be done.

Ada-11. Exceptions

Ada-11.1 Exception declarations

An exception declaration creates an EXCEPTION_SYM node.

```
label:   EXCEPTION_SYM
         (NAME NAME_NODE-label:)
```

Figure Ada-11-1: EXCEPTION_SYM node

Ada-11.2 Exception handlers

An exception handler looks almost like a case statment, except that the choices are restricted to being either exception names or others. Thus, separate operators are used to represent the exception handler.

```
label:   TREE_NODE
         (OP excp-case)
         (SUBNODES expr-label-sequence)

label:   TREE_NODE
         (OP excp-when)
         (SUBNODES EXCEPTION_SYM-label-sequence expr-label:)
```

Figure Ada-11-2: $TCOL_{Ada}$ representation for exception handler

The first subnode of an excp-when operator node may also be a TREE_NODE whose operator is "others". If any of the exceptions named by the exception node label sequence is the one which caused entry into the exception handler, the statements referred to in the last subnode are executed.

Ada-11.3 raise statements

```
raise exception_name;

label:    TREE_NODE
          (OP raise)
          (SUBNODES EXCEPTION_SYM-label:)
```

Figure Ada-11-3: $TCOL_{Ada}$ tree for raise statement

A raise statement with no exception named is legal only inside an exception handler, and it re-raises the exception which caused entry into the exception handler. This is identified in the TCOL tree by a separate operator, "re-raise".

```
declare
      -- declarations
begin
      -- statements
exception
      -- statments
      raise;
end;

label:    TREE_NODE
          (OP re-raise)
```

Figure Ada-11-4: TCOL tree for raise inside exception handler

---

Ada-11.3.1 Dynamic association of handlers with exceptions

Ada-11.4 Exceptions raised during tasking.

Ada-11.5 Raising an exception in another task

This has not yet been specified.

Ada-11.6 Supressing exceptions

Exceptions are suppressed by the SUPPRESS pragma. The scope of this pragma is the program unit in whose declarative part this pragma appears. Therefore, when elaborating the DECLARATION_INFO part of a program unit, the pragma can be found, and its applicability decided (i.e., whether or not the compiler chooses to honor it). Thus, there is no way the Front End can suppress the raise statement for a suppressed exception; that is something only later stages of the compiler can define.

Ada-12. Generic program units

In order to facilitate certain optimizations in simple compilers, a GENERIC_INFO node exists to link together all instances of generic procedure bodies. The complete specification of the GENERIC_INFO node is not finished.

```
label:   GENERIC_INFO
         (NAME NAME_NODE-label:)
         (INSTANCES SUBPROGRAM_SYM-label-sequence)
```

Figure Ada-12-1: GENERIC_INFO node

Ada-12.1 Generic Clauses

Ada-12.2 Generic Instantiation

Ada-13. Representation specifications

Ada-13.1 Packing Specifications

The appearance of a packing specification in the source text will cause the (PACKING YES) attribute value to be set. See section Ada-3.3.

Ada-13.2 Length Specifications

The appearance of a length specification in the source text will cause the LENGTH attribute value to be set. See section Ada-3.3.

Ada-13.3 Enumeration Type Representation

The appearance of an enumeration type representation in the source text will change the way in which the LITERALS of an ENUMERATION_SYM node are assigned values. See section Ada-3.5.1.

Ada-13.4 Record Type Representation

The presence of a record representation in the source text provides values for the LOCATION and ALIGNMENT attributes of the VARBL_SYM node for the record components. See section Ada-4.3.

Ada-13.5 Address Specifications

The appearance of an address specification in the source text has the following effects:

- For a variable, this causes the LOCATION attribute of the corresponding VARBL_SYM node to be set to the value of the location expression, expressed in bits. See section Ada-4.3. This must be a symbolic expression, because the Front End does not know how many bits comprise a storage unit.

- For the name of a subprogram, module or entry, this sets the LOCATION field in the SUBPROGRAM_SYM node; see section Ada-6.2.

Ada-13.5.1 Interrupts

Ada-13.6 Change of Representations

Ada-13.7 Configuration and Machine Dependent Constants

Ada-13.8 Machine Code Insertions

Ada-13.9 Interface to Other Languages

Ada-13.10 Unsafe Type Conversions

Ada-14. Input-output

Ada-1. Predefined language attributes

'ACCESS_SIZE

label: TREE_NODE
(OP access-size)
(SUBNODES TYPE_SYM-*label:*)

'ADDRESS

label: TREE_NODE
(OP address)
(SUBNODES expr-*label:*)

'BITS

label: TREE_NODE
(OP bits)
(SUBNODES TYPE_SYM-*label:*)

'CLOCK

label: TREE_NODE
(OP task-clock)
(SUBNODES expr-*label:*)

The subnode evaluates to the name of a task.

'COUNT

label: TREE_NODE
(OP entry-count)
(SUBNODES SUBPROGRAM-*label:*)

Subnode refers to an entry subprogram node.

'DELTA

label: TREE_SYM
(OP delta)
(SUBNODES TYPE_SYM-*label:*)

'DIGITS

label: TREE_SYM
(OP digits)
(SUBNODES TYPE_SYM-*label:*)

'EXPONENT_MAX

```
label:  TREE_SYM
        (OP exponent-max)
        (SUBNODES TYPE_SYM-label:)
```

'EXPONENT_MIN

```
label:  TREE_SYM
        (OP exponent-min)
        (SUBNODES TYPE_SYM-label:)
```

'FIRST

On scalar types.

```
label:  TREE_NODE
        (OP first)
        (SUBNODES TYPE_SYM-label:)
```

If the source language refers to an instance of a type then the Front End must supply the type of the instance as the operand.

'FIRST

On arrays, see 'FIRST(*i*).

'FIRST(i)

On arrays. If the parameter is omitted in the source text, the TCOL tree must have an explicit parameter of 1 supplied.

```
label:  TREE_NODE
        (OP first-bound)
        (SUBNODES TYPE_SYM-label: expr-label:)
```

If the source language refers to an instance of a type then the Front End must supply the type of the instance as the operand.

'FIRST_BIT

```
label:  TREE_NODE
        (OP first-bit)
        (SUBNODES VARBL_SYM-label:)
```

where the VARBL_SYM refers to a component VARBL in a record.

'INDEX

```
label:  TREE_NODE
        (OP task-index)
        (SUBNODES expr-label:)
```

The subnode evaluates to the name of a task.

'LARGE

```
label:  TREE_SYM
        (OP large)
        (SUBNODES TYPE_SYM-label:)
```

'LAST

```
label:  TREE_NODE
        (OP last)
        (SUBNODES TYPE_SYM-label:)
```

If the source language refers to an instance of a type then the Front End must supply the type of the instance as the operand.

'LAST

On arrays, see 'LAST(*I*).

'LAST(I)

On arrays. If the parameter is omitted in the source text, the TCOL tree must have an explicit parameter of 1 supplied.

```
label:  TREE_NODE
        (OP last-bound)
        (SUBNODES TYPE_SYM-label: expr-label:)
```

If the source language refers to an instance of a type then the Front End must supply the type of the instance as the operand.

'LAST_BIT

```
label:  TREE_NODE
        (OP last-bit)
        (SUBNODES VARBL_SYM-label:)
```

where the VARBL_SYM refers to a component VARBL in a record.

'LENGTH

See 'LENGTH(*i*).

'LENGTH(i)

On arrays. If the parameter is omitted in the source text, the TCOL tree must have an explicit parameter of 1 supplied.

```
label:    TREE_NODE
          (OP length)
          (SUBNODES TYPE_SYM-label: expr-label:)
```

If the source language refers to an instance of a type then the Front End must supply the type of the instance as the operand.

'ORD

```
label:    TREE_NODE
          (OP ord)
          (SUBNODES TYPE_SYM-label: expr-label:)
```

'POSITION

```
label:    TREE_NODE
          (OP position)
          (SUBNODES VARBL_SYM-label:)
```

where the VARBL_SYM refers to a component VARBL in a record.

'PRED

```
label:    TREE_NODE
          (OP pred)
          (SUBNODES TYPE_SYM-label: expr-label:)
```

'PRIORITY

```
label:    TREE_NODE
          (OP task-priority)
          (SUBNODES expr-label:)
```

The subnode evaluates to the name of a task.

'RADIX

label: TREE_SYM
(OP radix)
(SUBNODES TYPE_SYM-*label:*)

'REP

label: TREE_NODE
(OP rep)
(SUBNODES TYPE_SYM-*label:* expr-*label:*)

The DEFN attribute of the TREE_NODE refers to the function which will return the representation.

'SIZE

label: TREE_NODE
(OP size)
(SUBNODES TYPE_SYM-*label:*)

If the source language entity is the name of an instance of a type instead of the name of a type, then the Front End must supply the type reference.

'SMALL

label: TREE_SYM
(OP small)
(SUBNODES TYPE_SYM-*label:*)

'SUCC

label: TREE_NODE
(OP succ)
(SUBNODES TYPE_SYM-*label:* expr-*label:*)

'VAL

label: TREE_NODE
(OP val)
(SUBNODES TYPE_SYM-*label:* expr-*label:*)

The DEFN attribute of the TREE_NODE refers to the function which will return the value.

$TCOL_{Ada}$

Ada-2. Predefined Language Pragmas

$TCOL_{Ada}$

Ada-3. Predefined Language Environment

TCOL$_{Ada}$

Ada-4. Glossary

108 $TCOL_{Ada}$

$TCOL_{Ada}$

Ada-5. Syntax Summary

I. Summary of TCOL operators

&	Ada-3.6.3
*	Ada-4.5.5
**	Ada-4.5.6
/	Ada-4.5.5
+	Ada-4.5.4
-	Ada-4.5.4
=	Ada-4.5.2
/=	Ada-4.5.2
<	Ada-4.5.2
<=	Ada-4.5.2
>	Ada-4.5.2
>=	Ada-4.5.2
:=	Ada-5.1
:	Ada-5.b
abort	Ada-9.10
accept	Ada-9.5
access-size	Ada-A
address	Ada-A
agg-choice	Ada-3.6.2
aggregate	Ada-3.6.2
and	Ada-4.5.1
assert	Ada-5.9
bits	Ada-A

block	Ada-6.7
call	Ada-5.2
cand	Ada-5.4.1
case	Ada-5.5
component-select	Ada-4.1.2
cor	Ada-5.4.1
delay	Ada-9.6
delta	Ada-A
digits	Ada-A
elsif	Ada-5.4
entry-count	Ada-A
excp-case	Ada-11.2
excp-when	Ada-11.2
exit	Ada-5.c, Ada-5.7
exitlabel	Ada-5.c
exponent-max	Ada-A
exponent-min	Ada-A
first	Ada-A
first-bit	Ada-A
first-bound	Ada-A
for-down	Ada-5.6.c
for-up	Ada-5.6.c
function	Ada-6.4
goto	Ada-5.c, Ada-5.8

procedure	Ada-6.4
radix	Ada-A
raise	Ada-11.3
re-raise	Ada-11.3
rec-choice	Ada-3.7.3
record-aggregate	Ada-3.7.3
rep	Ada-A
return	Ada-5.3
return-value	Ada-5.3
size	Ada-A
slice	Ada-4.3.b, Ada-9.3
small	Ada-A
succ	Ada-A
task	Ada-6.4
task-clock	Ada-A
task-index	Ada-A
task-priority	Ada-A
U+	Ada-4.5.4
U-	Ada-4.5.4
val	Ada-A
value-procedure	Ada-6.4
when	Ada-5.5
while	Ada-5.6.b
xor	Ada-4.5.1

II. Summary of node types

```
label:   ACCESS_REP
         (ACCESS-OF TYPE_SYM-label:)


label:   ARRAY_REP
         (COMPONENT TYPE_SYM-label:)


label:   CONSTRAINT_REP
         (RANGE expr-label: expr-label:)
         (ACCURACY expr-label:)


label:   DECLARATION_INFO
         (SUBPROGRAMS SUBPROGRAM_SYM-label-sequence)
         (VARBLS VARBL_SYM-label-sequence)
         (TYPES TYPE_SYM-label-sequence)
         (EXCEPTIONS EXCEPTION_SYM-label-sequence)
         (PRAGMAS PRAGMA_SYM-label-sequence)
         (TASKS TASK_SYM-label-sequence)
         (PACKAGES PACKAGE_SYM-label-sequence)
         (ELABORATION_ORDER label-sequence)      ! to all nodes in
                                                 ! above attributes


label:   ENUMERATION_REP
         (LITERALS VARBL_SYM-label-sequence)


label:   EXCEPTION_SYM
         (NAME NAME_NODE-label:)


label:   GENERIC_INFO
         (NAME NAME_NODE-label:)
         (INSTANCES SUBPROGRAM_SYM-label-sequence)


label:   LABEL_SYM
         (NAME NAME_NODE-label:)
         (TREE expr-label:)


label:   LEAF_NODE
         (OP leaf)
         (SUBNODES VARBL-label:)
```

```
label:  LITERAL_REP
        (VALUE LG-literal)


label:  NAME_NODE
        (PNAME string)
        (NAMES label-sequence)                  ! TYPE_SYM,
                                                ! VARBL_SYM,
                                                ! EXCEPTION_SYM,
                                                ! LABEL_SYM,
                                                ! PRAGMA_SYM,
                                                ! PACKAGE_SYM,
                                                ! TASK_SYM,
                                                ! SUBPROGRAM_SYM


label:  PACKAGE_SYM
        (NAME NAME_NODE-label:)
```

The specification of the remainder of the PACKAGE_SYM node is not complete.

```
label:  PRAGMA_SYM
        (NAME NAME_NODE-label:)
        (ARGS label-sequence)
```

The exact specification of the ARGS attribute is not complete.

```
label:  RECORD_REP
        (FIELDS label-sequence)                 ! to VARBL_SYM nodes
                                                ! or TREE_NODE
                                                ! (op case) nodes


label:  SCALAR_REP
        (VARIETY FIXED | FLOAT | INTEGER | CHARACTER | BOOLEAN)


label:  SUBPROGRAM_SYM
        (NAME NAME_NODE-label:)
        (BODY expr-label-sequence)
        (RESULT TYPE_SYM-label:)
        (KIND  PROCEDURE | VALUE-PROCEDURE | FUNCTION
                         | ENTRY | TASK-BODY)
        (PARAMETERS VARBL_SYM-label-sequence)
        (LINKAGE LINKAGE-label:)
        (PRAGMAS PRAGMA_SYM-label-sequence)
        (DECLARATIONS DECLARATION_INFO-label:)
        (EXCEPTION expr-label:)
        (LOCATION expr-label:)
```

```
label:   TASK_SYM
         (DECLARATION DECLARATION_INFO-label:)
         (BODY SUBPROGRAM_SYM-label:)


label:   TREE_NODE
         (OP identifier)
         (DEFN label:)
         (SUBNODES expr-label-sequence)


label:   TYPE_SYM
         (KIND DECLARED | SUBTYPE | DERIVED | PREDEFINED)
         (NAME NAME_NODE-label:)
         (CONSTRAINT CONSTRAINT_REP-label-sequence)
         (PARENT TYPE_SYM-label:)
         (REP label:)                            ! ARRAY_REP,
                                                 ! RECORD_REP,
                                                 ! ENUMERATION_REP,
                                                 ! SCALAR_REP
         (PACKING YES | NO)                      ! Ada-13.2
         (LENGTH integer)                        ! Ada-13.2


label:   VARBL_SYM
         (NAME NAME_NODE-label:)
         (TYPE TYPE_SYM-label:)
         (CONSTANT NO | UNKNOWN | COMPILE | LINK | EXECUTION)
         (BINDING  IN | OUT | INOUT)             ! see text
         (LOCATION expr-label:)
         (LENGTH expr-label:)
         (ALIGNMENT expr-label)
         (INITIALIZE expr-label:)
```

References

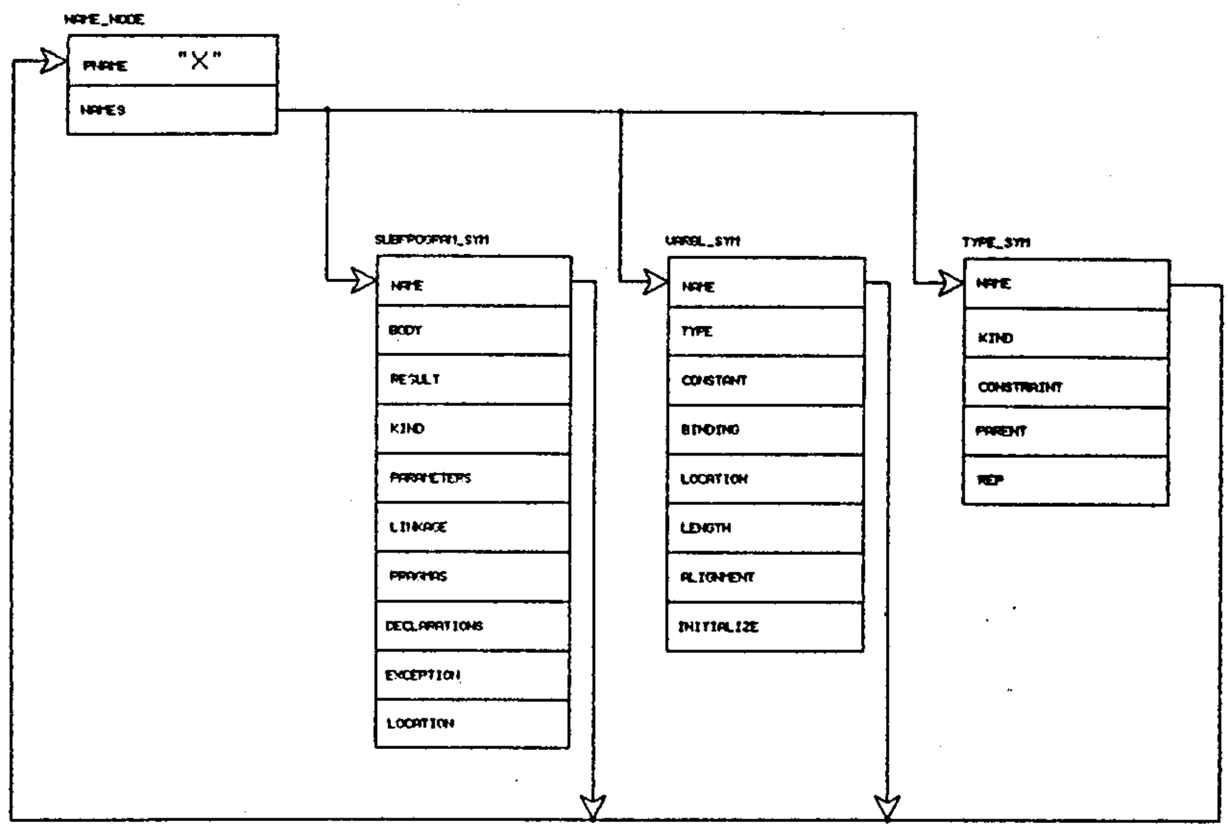
[1] R.G.G. Cattell.
Formalization and Automatic Derivation of Code Generators.
PhD thesis, Carnegie-Mellon University, April, 1978.

[2] 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.

[3] 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.

[4] J.M. Newcomer, R.G.G. Cattell, P.N. Hilfinger, S.O. Hobbs, B.W. Leverett, A.H. Reiner, B.R. Schatz, W.A. Wulf.
PQCC User's Manual.
Technical Report, Carnegie-Mellon University, Computer Science Department, May, 1979.

Index



COMPUTER SCIENCE ENGINEERING LAB

TITLE: Symbol Table Structure

PROJECT: TCOL.Ada Illustrations

DRAWN BY: Joseph M. Newcomer

CHECKED BY:

PAGE OF

DRAWING FILE: SYMTAB[C410JN11]

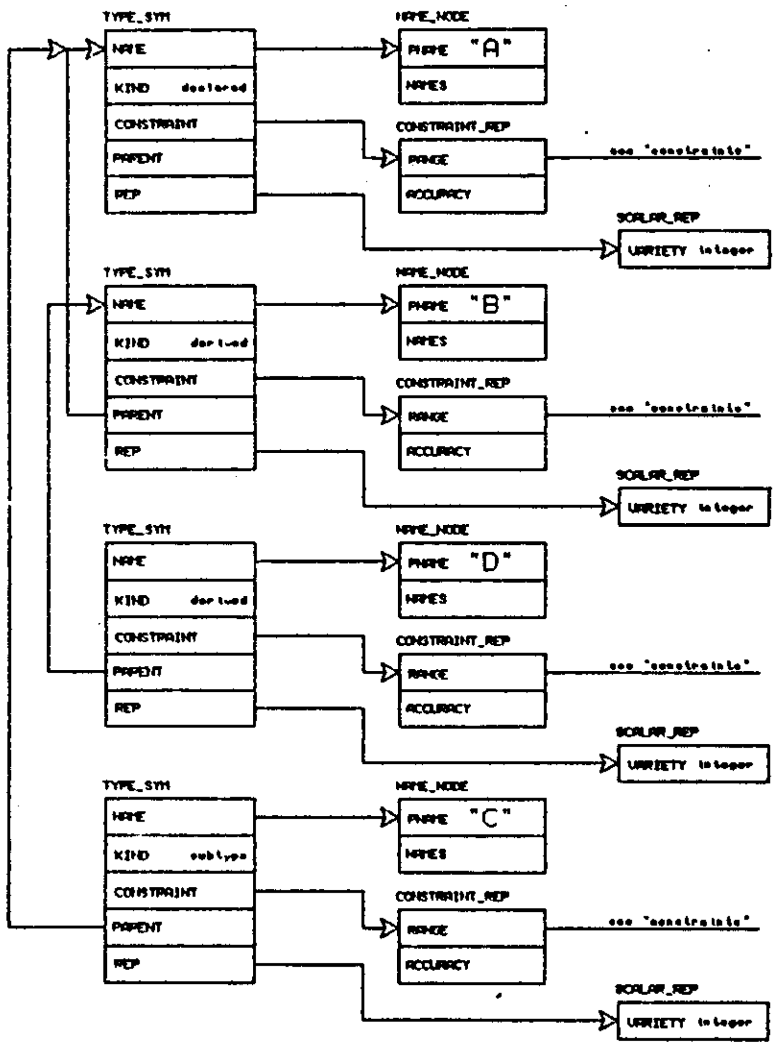
DRAWING NUMBER:

DATE: 21-JUN-79 01:09

CARNEGIE-MELLON UNIVERSITY

PITTSBURGH, PENNSYLVANIA 15213

```
type A is 1..1000;
type B is new A 10..50;
subtype C is A 50..100;
type D is new B 25..30;
```



COMPUTER SCIENCE ENGINEERING LAB

TITLE: Types, subtypes and derived types

PROJECT: TCOL.Ada Illustrations

DRAWN BY: Joseph M. Newcomer | CHECKED BY:

DRAWING FILE: TYPES[C410JN11] | DRAWING NUMBER:

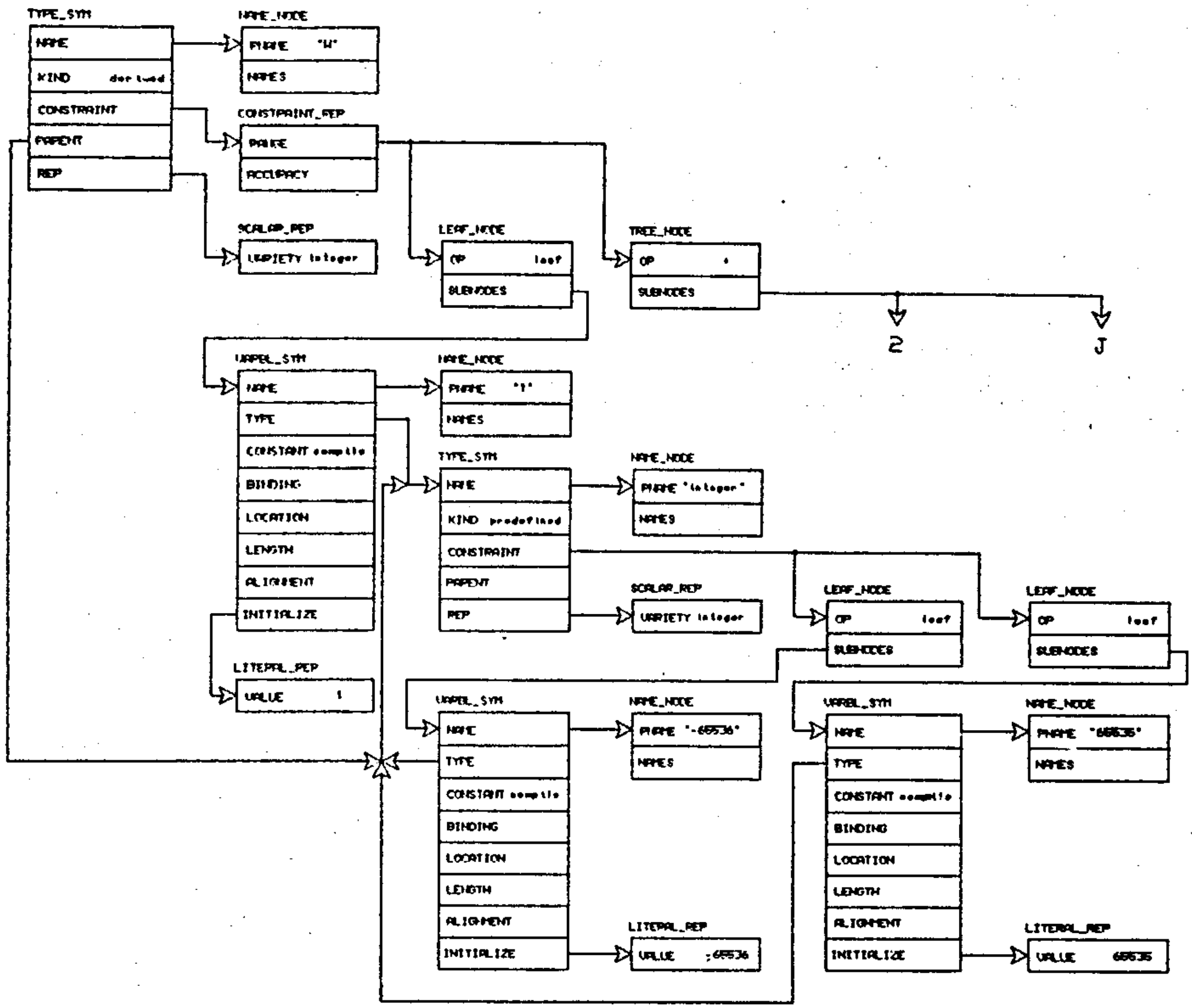
PAGE OF

DATE: 21-JUN-79 05:11

CARNEGIE-MELLON UNIVERSITY

PITTSBURGH, PENNSYLVANIA 15213

type W is new integer 1..2*J;



COMPUTER SCIENCE ENGINEERING LAB

TITLE: Constraints

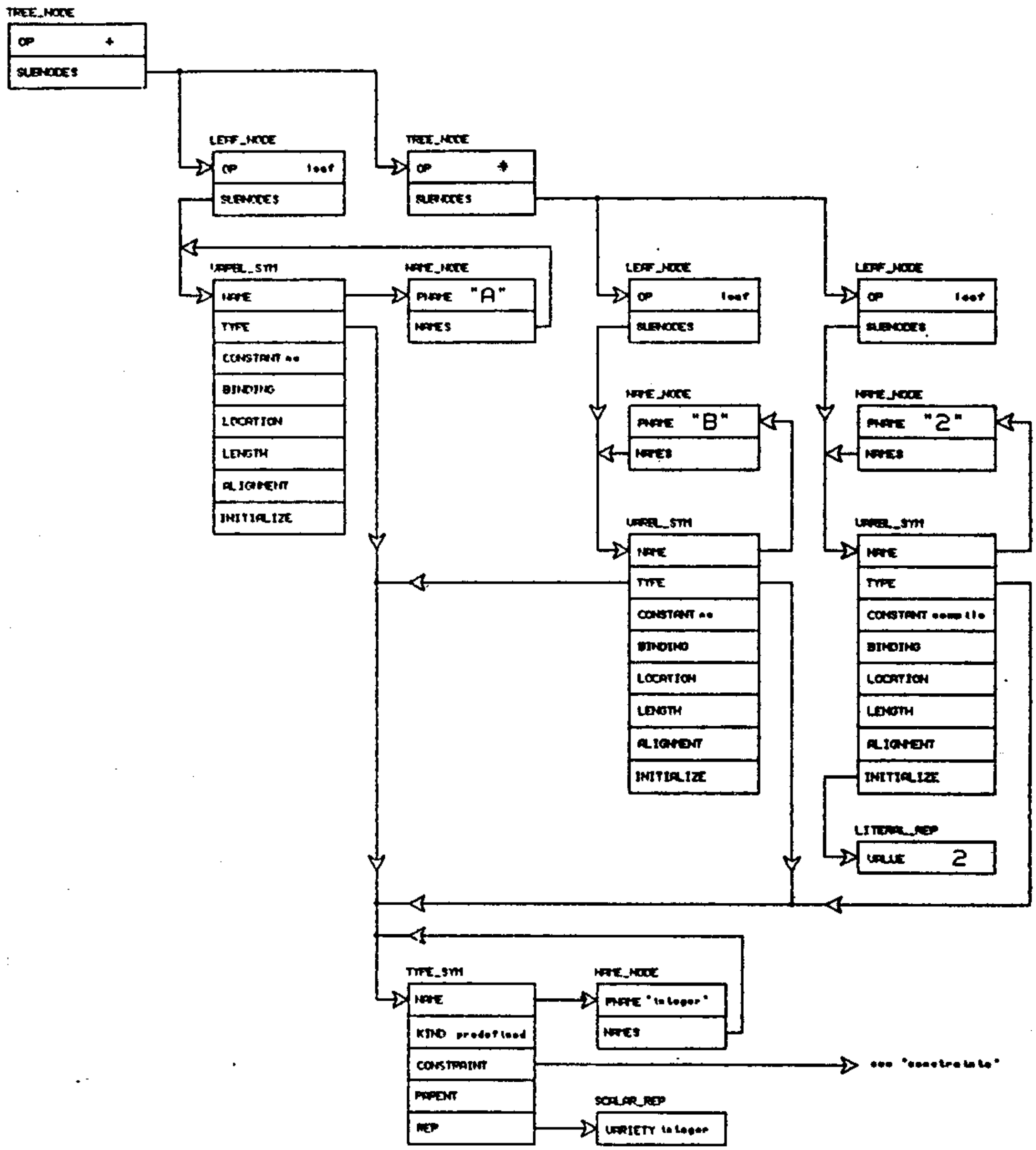
PROJECT: TCOL.Ada Illustrations

DRAWN BY: Joseph M. Newcomer | CHECKED BY: | PAGE OF

DRAWING FILE: CONST[C410JN11] | DRAWING NUMBER: | DATE: 20-JUN-79 23:44

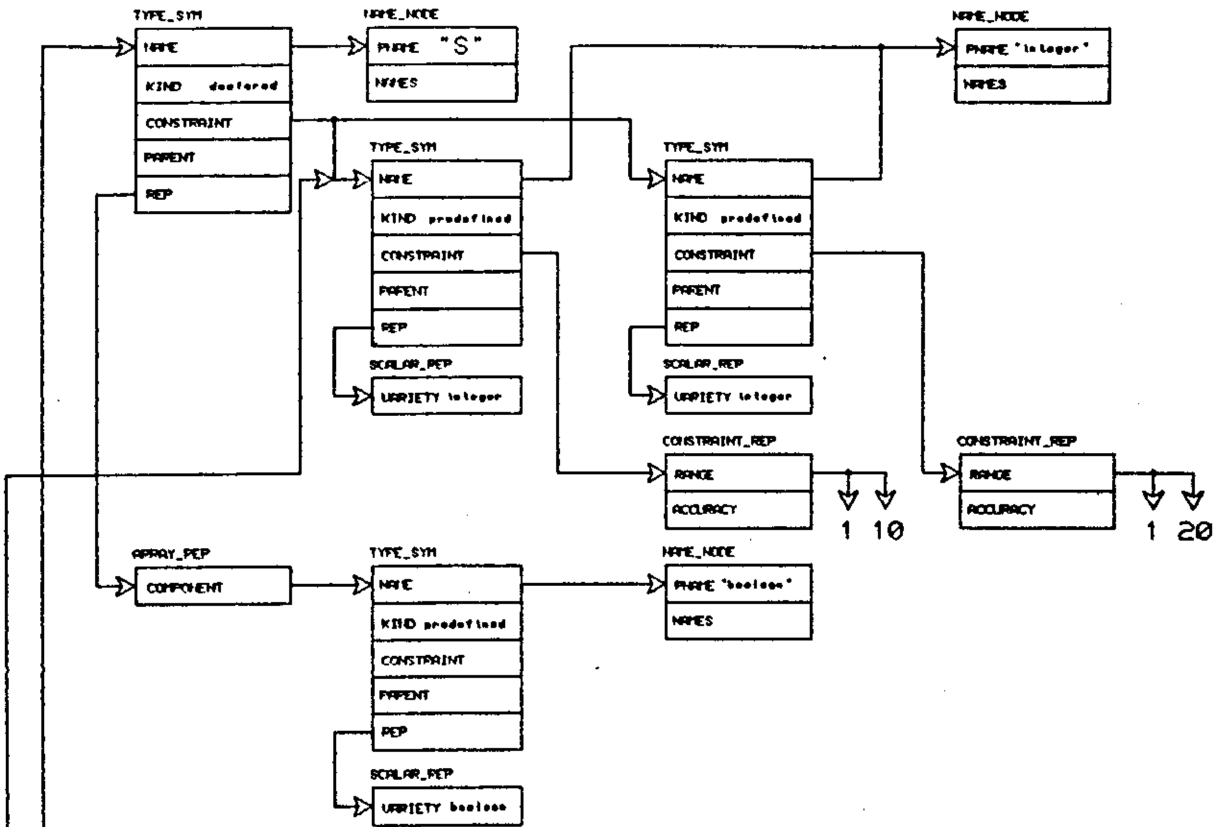
CARNEGIE-MELLON UNIVERSITY | PITTSBURGH, PENNSYLVANIA (G2T)

A+B*2

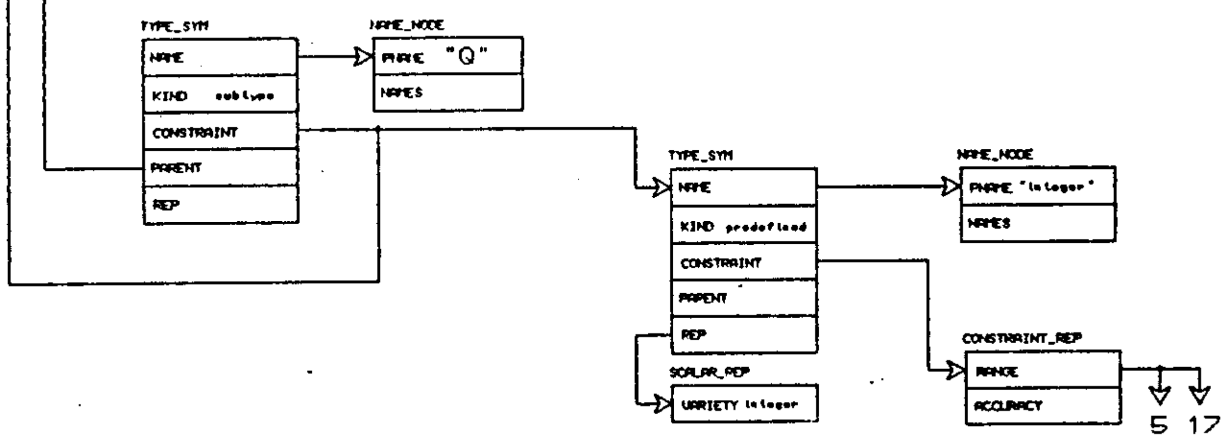


COMPUTER SCIENCE ENGINEERING LAB			
TITLE: Expression tree			
PROJECT: TCOL.Ada Illustrations			
DRAWN BY: Joseph M. Newcomer	CHECKED BY:	PAGE OF	
DRAWING FILE: EXPR[C410JN11]	DRAWING NUMBER:	DATE: 21-JUN-79 02:22	
CARNEGIE-MELLON UNIVERSITY		PITTSBURGH, PENNSYLVANIA 15213	

type S is array (1..10,1..20) of boolean;

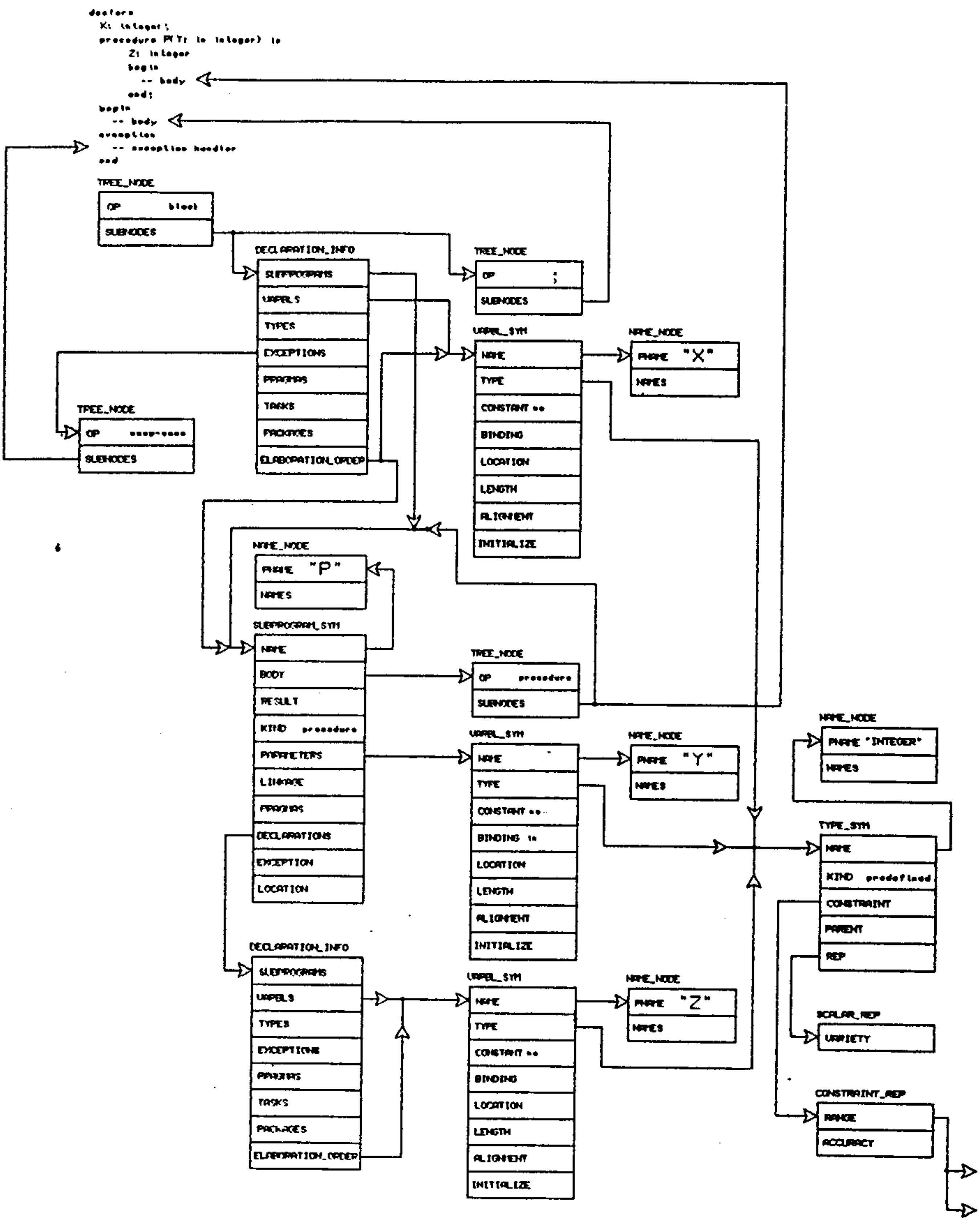


subtype Q is S(1..10,5..17) of boolean;



COMPUTER SCIENCE ENGINEERING LAB		
TITLE: Array representation		
PROJECT: TCOL.Ada Illustrations		
DRAWN BY: Joseph M. Newcomer	CHECKED BY:	PAGE OF
DRAWING FILE: ARRAY[C410JN11]	DRAWING NUMBER:	DATE: 21-JUN-79 06:46

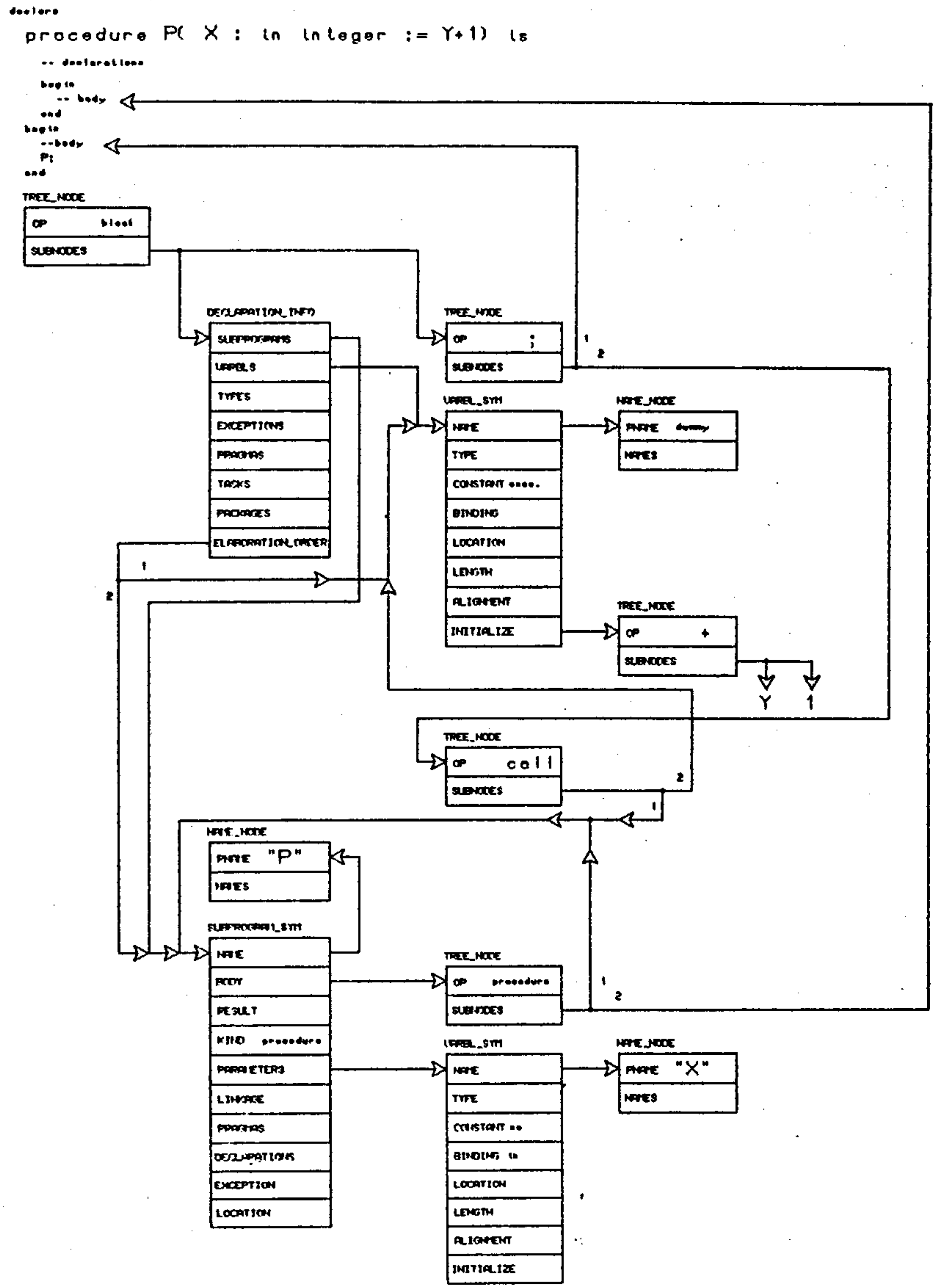
CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PENNSYLVANIA 15213



COMPUTER SCIENCE ENGINEERING LAB	
TITLE: Procedure declaration	
PROJECT: TCOL.Ada Illustrations	
DRAWN BY: Joseph M. Newcomer	CHECKED BY:
DRAWING FILE: PROC1[C410JN11]	DRAWING NUMBER:
PAGE OF	DATE: 21-JUN-79 03:09

CARNEGIE-MELLON UNIVERSITY

PITTSBURGH, PENNSYLVANIA (S213)



COMPUTER SCIENCE ENGINEERING LAB

TITLE: Default parameter representation

PROJECT: TCOL.Ada Illustrations

DRAWN BY: Joseph M. Newcomer

CHECKED BY:

DRAWING FILE: DEFAUL[C410JN11]

DRAWING NUMBER:

PAGE OF

DATE: 21-JUN-79 05:06

CARNEGIE-MELLON UNIVERSITY

PITTSBURGH, PENNSYLVANIA 15213