CMU-CS-79-112

510. 808
C28r
79-112
c.3

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

7 March 1979

Bruce R. Schatz
Bruce W. Leverett
Joseph M. Newcomer
Andrew H. Reiner
William A. Wulf

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

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Table of Contents

1. Introduction

This document describes $TCOL_{Ada}$, a possible intermediate representation for Ada -- the language being designed to meet the Steelman requirements. The purpose of this description is to provide enough information about this representation so that potential implementors of Ada can assess the feasibility of producing this representation at some intermediate stage of their compilers. Of course, Ada is not yet fully designed, and a choice between the two competing designs has not been made; thus, this document can only be considered tentative.[1] It should, however, provide enough of the "flavor" of $TCOL_{Ada}$ that an assessment can be made.

To understand what follows, it may be helpful to know a bit of the history of this representation. The "Production Quality Compiler-Compiler" (PQCC) project at Carnegie-Mellon University is attempting to automate the production of the optimization and code generation phases of a compiler. Starting with (only) formal descriptions of the source language and the target machine, PQCC is attempting to automatically generate "production quality" optimizers and code generators; our goal is to make these competitive with the best hand-generated systems currently available [2].

The PQC, "production quality compiler", produced by PQCC is heavily phase-structured; it consists of a linear sequence of phases, each of which performs a piece of the optimization/code generation process. TCOL was originally designed as the common representation input to the PQC, as well as for communication between the PQC phases [1]. In order to achieve the high standards set for the PQC, as well as to permit experimentation with the various PQC phases, it is necessary for this representation to have two properties:

- there must be a "visible" (ASCII) representation of TCOL -- this permits the constructors of the various phases to work independently and at their own pace, and

- the semantics of TCOL must be adaptable to the source language; that is, it must be possible to reflect the special properties of the source language types, operators, and control structures in TCOL. If this were not true, important optimization opportunities would be lost.

To satisfy these constraints, the PQCC project has defined a hierarchy of notations:

- LGN: LGN stands for "Linear Graph Notation", and is simply a linear, ASCII representation of arbitrary directed graphs.

[1] Interim drafts of the language reference manuals for Red and for Green were consulted in preparing this document. The Ada assumed within bears a fair resemblance to both of these.

- TCOL: TCOL stands for "Tree-structured Common Optimization Language", and is actually a family of languages -- one for each of the several source languages that we wish to deal with. The visible external representation of TCOL is written in LGN; that is, TCOL is simply an instance of LGN in which certain node attributes are presumed to exist, and in which the nodes happen to form a tree rather than a general graph.

- $\mathrm{TCOL}_{\mathrm{Ada}}$: $\mathrm{TCOL}_{\mathrm{Ada}}$ is the particular instance of TCOL which is tailored to the types, operators, and control constructs of Ada.

In the following sections we will describe LGN, the general properties of (any instance of) TCOL, and then a possible $\mathrm{TCOL}_{\mathrm{Ada}}$.

2. Abstract Representations

In this section we will discuss the general properties of LGN and TCOL. Since TCOL is a specialization of LGN, we will start with the latter first.

2.1 LGN: A Linear Graph Notation

LGN is a linearized, ASCII representation of an arbitrary directed graph with labelled arcs. Each node in the graph is presumed to be *labeled*, *typed*, and to have an arbitrary number of *attributes*. The attributes are *named*, and have *values*, but neither the names nor the possible values for a particular attribute are defined by LGN.

The definition of LGN, in BNF, is

```
<LG>    ::=    <empty>|<LG><node>
<node>  ::=     <label> <type> <attributes>
<label> ::=    <integer>:
<type>  ::=     <identifier>
<attributes>::= <empty>|<attribute><attributes>
<attribute> ::= (<at name> <at values>)
<at name>   ::= <identifier>
<at values> ::= <empty>|<at val><at values>
<at val>    ::= <label>|<identifier>|<string>|<number>
```

For example, the following is legal LGN:

```
1: PERSON  (SPOUSE 2: 3:) (SEX FEMALE) (CHILDREN 3:)
2: PERSON  (SPOUSE 1:) (SEX MALE) (CHILDREN 3:) (KILLED-BY 4:)
3: PERSON  (SPOUSE 1:) (SEX MALE) (NAME "Oedipus") (PARENTS 1: 2:)
4: DISCUS  (FEET-THROWN 100)  (THROWN-BY 3:)
```

This bit of LGN defines a graph with four nodes (labeled 1: - 4:). Three of the nodes have type PERSON and one has type DISCUS. The nodes are connected together in various ways by attributes whose values are labels. For example, the SPOUSE attribute of node 1: points to nodes 2: and 3: , indicating that PERSON 1:'s spouses are PERSONs 2: and 3:. Note that lists of values are permited for any attribute. Attributes which can only take on a small number of values have those values denoted by unique identifiers. For example, the SEX attribute may take on only the value FEMALE or the value MALE. Attributes which can take on many (infinite number of) values have those denoted by quoted strings or by numbers. For example, the NAME attribute in node 3: has value "Oedipus" and the FEET-THROWN attribute in node 4: has the value 100.

The appendix to this report contains several examples of TCOL_{Ada} in LGN format. Hopefully these examples will be self explanatory.[1]

2.2 TCOL: A Tree-Structured Intermediate Language

As discussed previously, TCOL is actually a family of languages -- one for each of the several source languages for which we plan a PQCC-produced compiler. Thus TCOL can be viewed as the union of the properties of the various language-specific family members. Remember that the visible representation of TCOL is a restriction of LGN.

The TCOL representation of a program is an abstract program syntax tree. It consists of a sequence of interconnected nodes with named operators in the interior nodes and various attributes attached to the leaf nodes. The tree structuring is constant over all TCOLs while the operators and attributes vary for each source language. The set of operators represents the operations of a virtual machine intermediate between the source and machine languages. In level, TCOL resembles an abstract programming language, i.e. the set of operators is much closer to the particular source language than to the machine code.

2.2.1 Common nodes and attributes

The fact that the tree structure is constant over all TCOL family members means that there is a specific set of node types and attributes common to all. These will now be discussed.

Each TCOL tree has a distinguished *root node*. Each interior node, termed a *tree node*, represents an expression consisting of an operator and one or more operands which are represented by the subtrees pointed to by the SUBNODES attribute. The leaves of the tree are termed *leaf nodes*. The leaf nodes themselves are only place holders, however, and point to *symbol nodes*, representing symbol table entries, or to *literal nodes*, representing literals (constants) from the source program. These latter two types of nodes contain the actual values of the leaves of the tree.

Thus the TCOL tree structure is represented by an initial root node and a sequence of nodes with the following types and attributes. Each node type is given followed by its attributes (in LGN format). Here upper case will denote attribute names and lower case will denote value types (however, case is not significant in LGN). Thus (OP identifier) means the OP attribute is specified by identifier values.

[1] We should note that the reason for defining LGN in addition to, and more general than, TCOL is that PQCC must also provide visible representations for control flow graphs, internal tables, and other structures which may not be trees and which require a different (and possibly richer) set of node types and attributes.

TREE	defines operator (interior of abstract syntax tree).	
	`(OP identifier)`	name of TCOL operator.
	`(ANCESTOR label)`	
	`(SUBNODES label-list)`	
LEAF	defines operand (leaves of abstract syntax tree).	
	`(OP rv)`	where rv = SYMBOL,LITERAL.
	`(ANCESTOR label)`	
	`(VALUE label)`	points to symbol or literal node.
SYMBOL	defines symbol ("storage location").	
	`(NAME string)`	name of symbol table entry.
LITERAL	defines literal ("value location").	
	`(VALUE value)`	permissible values depend on language datatypes.

For each specific TCOL (specific source language), there are additional attributes and possibly additional node types. In particular, the source language datatypes must be reflected by attributes in the Symbol and Literal nodes, and the set of Tree operators must reflect the semantics of the various language constructs.

2.2.2 The Front End

The program which transforms source language into TCOL (termed the *front end*) must do more than lexical and syntactic analysis. This is because the TCOL "parse tree" is actually an abstract syntax tree which also hides some of the semantics of the source language. Of this, some is recorded in attributes attached to the nodes (e.g. symbol table references in Symbol nodes) while some is no longer necessary by TCOL time (e.g. compile-time coercions). In particular, the front end is assumed to be responsible for:

- name scoping and symbol table generation
- compile-time type coercion and type checking
- elaboration and use checking of encapsulated objects

- macro and generic expansion
- compile-time diagnostics and pragmats
- source file inclusions ("require"s and library manipulation)
- conditional compilation.

Several implications of these should be noted.

- The coercions include deferencing (using the explicit "contents-of" operator), expanding references to structured variables (e.g. arrays, records) into address computations for individual memory accesses, and expressing all references to user or standard prelude types in terms of operations on their underlying built-in types.
- The operands point to specific storage table entries so that all scoping references have been factored out (i.e. symbol table references are given in Symbol and Literal nodes as needed).
- There is a distinct operator for each applicable type, e.g. plusInteger,plusReal, plusLongReal are as distinct as plus and times.

In essence, this document presents a front-end's-eye view of TCOL; details internal to PQCC are suppressed.

3. TCOL$_{Ada}$

The remainder of this document will define some specifics for TCOL$_{Ada}$, keeping in mind that these are necessarily tentative until the details of Ada itself are finalized.

As TCOL$_{Ada}$ is a particular member of the TCOL family, it uses all of the common types of nodes and attributes which define the tree structure, as described in section 2.2.1. However, it adds a number of attributes to Symbol and Literal nodes which specify the data type and lifetime information for Ada. It also includes a number of tree operators meant to provide an abstract machine for the semantics of Ada. (It should be noted that many of these operators are common to other algebraic languages.)

The remainder of this section will give the set of node types used in TCOL$_{Ada}$. Then the attributes for datatypes will be enumerated, the tree (language construct) operators given, and finally, several examples will be presented.

3.1 Types of nodes

For TCOL$_{Ada}$, no additional node types beyond those common to all TCOLs will be proposed. Several additional attributes are necessary in Symbol and Literal nodes to specify typing and lifetime information. The list of node types and attributes here includes (reiterates) the common tree structure (as discussed in section 2.2.1) while the attributes necessary for data typing are given in the next section. The notation is the same as previously: the node type is given followed by its attributes (in LGN format where upper case denotes attribute names and lower case denotes value types). "rv" stands for "range of values" and is used when permissible values include only a few identifiers.

TREE

```
(OP identifier)  name of TCOL operator.
(ANCESTOR label)
(SUBNODES label-list)
```

LEAF

```
(OP rv)            where rv = SYMBOL, LITERAL .
(ANCESTOR label)
(VALUE label)      points to a Symbol or Literal node.
```

SYMBOL

(NAME string)	name of symbol table entry.
(TYPE type)	one of basic types defined in next section.
(LIFE rv)	where rv = STATIC,STACK,HEAP (lifetime type).
(GLOBAL rv)	where rv = YES,NO (whether externally visible).

LITERAL

```
(VALUE value)
(TYPE type)
```

Any scalar type is permissible in a Literal. The value is then a single value of that type. The one exception is that types INT, FIXED, and FLOAT are allowed to specify (VALUE ENUM) which indicates that the literal consists of an enumeration of all values in the subtype range (e.g. in a FOR statement). Type ARRAY is also permissible in a Literal in which case the value is a list of the constant values of the array elements.

For Symbol and Literal nodes, there are additional type-dependent attribute fields which must be merged into the ones specified above. Thus the fields in any such node consist of the standard ones above and the specific ones for the particular type. Unneeded attributes may be left undefined. The next section describes the type-specific attributes.

It should be noted that the intermediate language TCOL does not define the semantics of a programming language, but merely reflects it (in the choice of operators and attribute types, for example). There must, however, be at least an implicit agreement on the semantics between the designers of Ada (and its language description) and the designers of TCOL_{Ada}.

4. Language Datatypes

The basic types listed here are the only ones which appear in TCOL$_{Ada}$ trees (in the Type field of Symbol and Literal nodes). The additional attributes which must be added to Symbol or Literal nodes for each type are given.

4.1 Scalar types

INT	(RANGE int int)
FIXED	(RANGE fixed fixed) (DELTA fixed)
FLOAT	(RANGE float float) (PRECISION int)
BOOL	
CHAR	
STRING	(LENGTH int) maximum length of multicharacter string.
PTR	(POINTS-TO type)
BUILTIN	value is a standard identifier (for inquiries such as PRECISION).

4.2 Structured types

ARRAY	(ELTTYPE type) (BOUNDS lb1 ub1 lb2 ub2 ...) where all bounds are integers. For dynamic arrays, the bounds are pointers to (labels of) Symbol nodes. The initialization code for a dynamic array declaration should compute each bounds expression and place the value in a dummy variable. These variables are then represented by TCOL Symbol nodes and pointed to in the BOUNDS attribute.
RECORD	(FIELDS label-list) list of pointers to the Symbol nodes which form the subfields.
SET	(ELTTYPE type) (SIZE int)
UNION	(FIELDS label-list) A Symbol node of this type represents a single storage location. Its subfields point to dummy Symbol nodes which specify the type of that location for the various selections of the discriminated union.

4.3 Linking types

PARAMETER	(PARMTYPE type) (PARMNUM int) (BY rv) where rv = VAL, REF, RESULT, REFIN (which passing mechanism).
ROUTINE	(EFFECT rv) where rv= PURE, IMPURE (for names of routines).
TASK	a variable of this type has value = set of active processes for this task.

The front end is assumed to change enumerated type references into references to small integers so that each enumerated type becomes some subtype of **INT**. The same should occur with standard enumerations, such as the list of exceptional conditions.

Any fields needed for attribute inquiries (such as **FIRST** for type **ARRAY** or **TAG** for type **UNION**) should also be included above.

The remainder of this document describes the tree operators and gives several examples of their use.

5. Operators on basic types

Notation: e_0 - e_n are the 0th - nth operands of the operator. They are assumed to be expressions unless stated otherwise.

The list of operators here specifies a generic set. However, after the coercions done by the front end, the actual trees will contain type-specific operators. Thus for +, the actual set of operators would include +INT, +FIXED, and +FLOAT. Similarly, the logical comparison operators, such as =, may have both a signed and an unsigned version.

5.1 Arithmetic

+	addition: $e_0 + e_1$
-	subtraction: $e_0 - e_1$
/	division: e_0 / e_1
*	multiplication: $e_0 * e_1$
**	exponentiation: $e_0 ** e_1$
mod	modulo: e_0 **mod** e_1
--	unary minus: $-e_0$.
++	unary plus: $+e_0$.
abs	absolute value: **abs** (e_0)
succ	next higher value: **succ** (e_0)
pred	next lower value: **pred** (e_0)

5.2 Logical

These return type boolean.

<	less than: $e_0 < e_1$.
>	greater than: $e_0 > e_1$.
<=	less than or equal to: $e_0 <= e_1$.
>=	greater than or equal to: $e_0 >= e_1$.
=	equal to: $e_0 = e_1$.
~=	not equal to: $e_0 \sim= e_1$.
and	bitwise and: e_0 **and** e_1.

or	bitwise or: e_0 **or** e_1.
xor	bitwise exclusive or: e_0 **xor** e_1.
not	bitwise not: **not** e_0.
member	returns true iff e_0 is a member of set e_1.

5.3 String

concat	concatenation: e_0 concat e_1.
substring	substring of e_0 from e_1 to e_2.
length	current length of e_0.

5.4 Pointer

new	return new variable of scalar type e_0: **new**(e_0).
free	deallocate heap variable pointed to by ptr e_0: **free**(e_0).

5.5 Structured type

It is assumed that all references to these have been decomposed into individual element references by TCOL time. Thus the only operators provided here are those for extraction.

arrayelt	for array expression e0, return pointer to e0[e_1,e_2...].
subfield	for record e_0, return pointer to subfield named e_1.
discriminate	for variant union e_0, choose type in Symbol node pointed to by e_1.

5.6 Value

:=	assignment to variable: e_0 := e_1.
contents	contents-of (value of): contents e_0 .
inquiry	for symbol e_0, return value of attribute identified by e_1. Permissible inquiries depend on the type of the Symbol node pointed to by e_0. e_1 points to a Literal node of type **BUILTIN**.

6. Control operators

6.1 Sequencing & jumps

;	sequencing. arbitrary numbers of operands.
exit	structured jump from inside labelled construct to after it: **exit** e_0 .
goto	jump to label: **goto** e_0.
label	e_0 is label on statement e_1: "$e_0 : e_1$" .

6.2 Conditionals

if	**if** e_0 **then** e_1 **else** e_2 **endif** .
case	**case** e_0 **when** <cond> => <expr> **when** <cond> => <expr> ... **else** e_n **endcase** is a subtree with n+1 operands with the first being the selection expression, the last being the else expression, and the others (i.e. $e_1...e_{n-1}$) being a subtree with root operator acase.
acase	used only within a case expression tree. acase is an n-ary operator representing "when <cond> => <expr>" with the first operand being the executed expression and the second and subsequent operands representing the guarding conditions. Each condition is either an expression (represented by the unary operator econd whose operand is the expression's subtree) or a range (represented by the binary operator rcond whose operands are the (constant) lower and upper bound).

6.3 Repetition

while	**while** e_0 **repeat** e_1 **endrepeat**.
for	**for** $e_0 : e_1$ **repeat** e_2 **endrepeat** . e_0 is a variable, e_2 is an expression, and e_1 is a subtype (subrange) of type int, fixed, or float, or is an enumerated type. The effect is "repeat e_2 varying e_0 over all values of e_1".

6.4 Routine declarations and calls

function	pure routine declaration: **function** e_0 (e_3,...,e_n): e_1; e_2 . e_0 is the name (points to a Symbol node of type "routine"), e_1 is the returntype, e_2 is the body expression, $e_3...e_n$ point to Symbol nodes of type "parameter".

procedure	impure routine declaration: **procedure** e_0 (e_2...e_n); e_1 . name e_0, body e_1, parameters e_2...e_n .
return	return from a function with value e_0: **return** e_0.
call	routine call: e_0(e_1..e_n). e_0 references a Symbol node of type "routine", e_1..e_n point to expression subtrees representing the arguments.

7. Miscellaneous Operators

These are somewhat minimal and speculative.

7.1 Exception handling

Exception conditions are represented by small integers. Thus e_0 in **raise** and $e_1, e_3, \ldots$ in **guard** are Leaf nodes whose value field points to a Literal node which gives the exception number.

raise	signal condition: **raise** e_0.
guard	**guard** e_0 **by when** e_1 => e_2 **when** e_3 => e4 ... **else** e_{2n+1} **endguard.**

7.2 Parallel processing

fork	for task e_0 activate processes $e_1 \ldots e_n$ (allow them to run concurrently with other activated processes). $e_1 .. e_n$ are pointers to Symbol nodes of type "routine".
join	for task e_0 deactivate processes $e_1 \ldots e_n$ (e.g. when abort,terminate, or reach block end).
wait	selective wait: **wait** e_0 => e_1, e_2 => e_3 ... **endwait.**

7.3 Input/output

read	from device e_0, read into variables $e_1 \ldots e_n$ (free-format).
write	to device e_0, write from variables $e_1 \ldots e_n$ (free-format).

8. Coda

8.1 Omissions

Due to the preliminary nature of this document and of Ada, a number of issues have not been dealt with in sufficient depth. Some of these are listed below.

- Machine dependent support. This includes access to the interrupt structure and linkage to assembly language routines.

- Runtime system support. This includes specification of the stack frame and display, dynamic storage allocation (e.g. heaps and garbage collection), and runtime checking of bounds and types.

- Parallel processing support. This includes definition and naming of processes, types of activation variables for tasks, and scheduling.

- Fixed point support. This includes scaling, representation of literals, and whether **INT** should be a subtype of **FIXED** rather than a separate type.

- Encoding pragmat information about the representation of data structures in the TCOL Symbol nodes.

- High-level input/output support including file manipulation.

- The representation of load-time constants.

8.2 Acknowledgements

Paul Hilfinger provided helpful comments on several drafts of this paper. The first author (B.R.S.) was supported by a graduate fellowship from the National Science Foundation.

8.3 References

[1] R.G.G. Cattell, *Formalization and automatic derivation of code generators*, Rep. CMU-CS-78-115, (Carnegie-Mellon University, Computer Science Department, April 1978).

[2] 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*, Rep. CMU-CS-79-105, (Carnegie-Mellon University, Computer Science Department, February 1979).

A. Examples

This appendix gives a few examples of the TCOL_{Ada} trees for various Ada source constructs. In each example, the Ada source is given (with hypothetical syntax) followed by the TCOL tree in LGN notation.

A.1 Statements

```
BEGIN
  VAR c,a,b : INT (1..10),
      d : FLOAT (5,1E-1..1E3);
  c := 7;    a := b + c;   d := 0.31416E1
END

1: SYMBOL  (NAME "C") (TYPE INT) (LIFE STACK) (GLOBAL NO)
           (RANGE 1 10)
2: SYMBOL  (NAME "A") (TYPE INT) (LIFE STACK) (GLOBAL NO)
           (RANGE 1 10)
3: SYMBOL  (NAME "B") (TYPE INT) (LIFE STACK) (GLOBAL NO)
           (RANGE 1 10)
4: SYMBOL  (NAME "D") (TYPE FLOAT) (LIFE STACK) (GLOBAL NO)
           (RANGE 1E-1 1E3) (PRECISION 5)
5: LITERAL (VALUE 7) (TYPE INT)
6: LITERAL (VALUE 0.31416E1) (TYPE FLOAT) (PRECISION 5)

20: TREE  (OP ;) (ANCESTOR 0:) (SUBNODES 21: 24: 31:)
21: TREE  (OP :=) (ANCESTOR 20:) (SUBNODES 22: 23:)
22: LEAF  (OP SYMBOL) (ANCESTOR 21:) (VALUE 1:)
23: LEAF  (OP LITERAL) (ANCESTOR 21:) (VALUE 5:)
24: TREE  (OP :=) (ANCESTOR 20:) (SUBNODES 25: 26:)
25: LEAF  (OP SYMBOL) (ANCESTOR 24:) (VALUE 2:)
26: TREE  (OP +INT) (ANCESTOR 24:) (SUBNODES 27: 29:)
27: TREE  (OP CONTENTS) (ANCESTOR 26:) (SUBNODES 28:)
28: LEAF  (OP SYMBOL) (ANCESTOR 27:) (VALUE 3:)
29: TREE  (OP CONTENTS) (ANCESTOR 26:) (SUBNODES 30:)
30: LEAF  (OP SYMBOL) (ANCESTOR 29:) (VALUE 1:)
31: TREE  (OP :=) (ANCESTOR 20:) (SUBNODES 32: 33:)
32: LEAF  (OP SYMBOL) (ANCESTOR 31:) (VALUE 4:)
33: LEAF  (OP LITERAL) (ANCESTOR 31:) (VALUE 6:)
```

Notes: The Symbol and Literal nodes contain the symbol table information; they are referenced in the Leaf nodes. The Symbol nodes contain the basic attributes on first line and the type-specific (e.g. RANGE for INT) attributes on second line. The RANGE attribute is omitted for the Literals (is meaningless).

The semicolon treenode (20:) captures the sequencing of top-level statements in the program. Its operands are the assignments (21: 24: 31:). The LGN nodes are shown here in depth-first treewalk order.

The CONTENTS operator takes the values of variables on the right hand side of an assignment.

The plus operator in node 26: has been coerced into a type-specific +INT by the front end.

The 0: in the Ancestor field of the first treenode (20:) points to the rootnode (or to the next larger expression subtree).

A.2 Array access in a For loop

```
BEGIN
  TYPE color = ENUM [red,green,blue];
  VAR  I : INT(1..1000),
       a : ARRAY [1..3] of INT(1..10);
  FOR I : color  REPEAT  a[I] := 2  END REPEAT;
END

1: SYMBOL  (NAME "I") (TYPE INT) (LIFE STACK) (GLOBAL NO)
           (RANGE 1 1000)
2: SYMBOL  (NAME "A") (TYPE ARRAY) (LIFE STACK) (GLOBAL NO)
           (ELTTYPE INT) (BOUNDS 1 3)
           (RANGE 1 10)
4: LITERAL (VALUE 2) (TYPE INT)

20: TREE   (OP FOR) (ANCESTOR 0:) (SUBNODES 21: 22: 24:)
21: LEAF   (OP SYMBOL) (ANCESTOR 20:) (VALUE 1:)
22: LEAF   (OP LITERAL) (ANCESTOR 20:) (VALUE 23:)
23: LITERAL  (VALUE ENUM) (TYPE INT) (RANGE 1 3)
24: TREE   (OP :=) (ANCESTOR 20:) (SUBNODES 25: 29:)
25: TREE   (OP ARRAYELT) (ANCESTOR 24:) (SUBNODES 26: 27:)
26: LEAF   (OP SYMBOL) (ANCESTOR 25:) (VALUE 2:)
27: TREE   (OP CONTENTS) (ANCESTOR 25:) (SUBNODES 28:)
28: LEAF   (OP SYMBOL) (ANCESTOR 27:) (VALUE 1:)
29: LEAF   (OP LITERAL) (ANCESTOR 24:) (VALUE 4:)
```

Notes: Symbol "A" has one merge of type-specific attributes for the (TYPE ARRAY) and another for the (ELTTYPE INT). The special Literal 23: is generated as an INT range for the loop variable to vary over (note the reference to an enumerated type has become a reference to a small int subtype).

A.3 Routine declaration

```
p : PROCEDURE (x : INT(1..10), VAR y : BOOL);
     ( y := IF x = 1  THEN TRUE  ELSE FALSE  ENDIF  );

1: SYMBOL  (NAME "P") (TYPE ROUTINE) (LIFE STACK) (GLOBAL YES)
           (EFFECT IMPURE)
2: SYMBOL  (NAME "X") (TYPE PARAMETER) (LIFE STACK) (GLOBAL NO)
           (PARMTYPE INT) (PARMNUM 1) (BY VAL)
           (RANGE 1 10)
3: SYMBOL  (NAME "Y") (TYPE PARAMETER) (LIFE STACK) (GLOBAL NO)
           (PARMTYPE BOOL) (PARMNUM 2) (BY REF)
4: LITERAL (VALUE 1) (TYPE INT)
5: LITERAL (VALUE TRUE) (TYPE BOOL)
6: LITERAL (VALUE FALSE) (TYPE BOOL)

20: TREE  (OP PROCEDURE) (ANCESTOR 0:)
          (SUBNODES 21: 22: 31: 32:)
21: LEAF  (OP SYMBOL) (ANCESTOR 20:) (VALUE 1:)
22: TREE  (OP :=) (ANCESTOR 20:) (SUBNODES 23: 24:)
23: LEAF  (OP SYMBOL) (ANCESTOR 22:) (VALUE 3:)
24: TREE  (OP IF) (ANCESTOR 20:) (SUBNODES 25: 29: 30:)
25: TREE  (OP =) (ANCESTOR 24:) (SUBNODES 26: 28:)
26: TREE  (OP CONTENTS) (ANCESTOR 25:) (SUBNODES 27:)
27: LEAF  (OP SYMBOL) (ANCESTOR 26:) (VALUE 2:)
28: LEAF  (OP LITERAL) (ANCESTOR 25:) (VALUE 4:)
29: LEAF  (OP LITERAL) (ANCESTOR 24:) (VALUE 5:)
30: LEAF  (OP LITERAL) (ANCESTOR 24:) (VALUE 6:)
31: LEAF  (OP SYMBOL) (ANCESTOR 20:) (VALUE 2:)
32: LEAF  (OP SYMBOL) (ANCESTOR 20:) (VALUE 3:)
```

Notes: For the procedure, the body starts at 22: while the parameters start at 31: . The IF has three parts: condition (25:), then (29:), and else (30:).

Variables of type BOOL (e.g. "Y") have no type-specific fields.

This procedure is GLOBAL (i.e. it is known to the linker and can be referenced externally).

A.4 Records

```
BEGIN
  VAR book : RECORD
       (author,title : STRING(20),
        date : INT(1500,,2000));
  book.author :=  "Bram Stoker";
  book.title := "Dracula";
  book.date := 1897
END
```

```
1: SYMBOL (NAME "BOOK") (TYPE RECORD) (LIFE STACK) (GLOBAL NO)
          (FIELDS 2: 3: 4:)
2: SYMBOL (NAME "AUTHOR") (TYPE STRING) (LIFE STACK) (GLOBAL NO)
          (LENGTH 20)
3: SYMBOL (NAME "TITLE") (TYPE STRING) (LIFE STACK) (GLOBAL NO)
          (LENGTH 20)
4: SYMBOL (NAME "DATE") (TYPE INT) (LIFE STACK) (GLOBAL NO)
          (RANGE 1500 2000)
5: LITERAL (VALUE "Bram Stoker") (TYPE STRING) (LENGTH 11)
6: LITERAL (VALUE "Dracula") (TYPE STRING) (LENGTH 7)
7: LITERAL (VALUE 1897) (TYPE INT)

20: TREE  (OP ;) (ANCESTOR 0:) (SUBNODES 21: 26: 31:)
21: TREE  (OP :=) (ANCESTOR 20:) (SUBNODES 22: 25:)
22: TREE  (OP SUBFIELD) (ANCESTOR 21:) (SUBNODES 23: 24:)
23: LEAF  (OP SYMBOL) (ANCESTOR 22:) (VALUE 1:)
24: LEAF  (OP SYMBOL) (ANCESTOR 22:) (VALUE 2:)
25: LEAF  (OP LITERAL) (ANCESTOR 21:) (VALUE 5:)
```

... the other two record accesses are similar to 21: - 25: .

Notes: TCOL does not specify the physical representation of records. Thus, for example, a string could either be a block of storage within the record or a pointer to string space elsewhere.

A.5 Unions within Records

```
BEGIN
  VAR  num : RECORD
        (..., ri: UNION [ r : FLOAT (3, 1E0..1E1),
                          i: INT (1..10) ], ... );

  num.ri.i  :=  num.ri.i + 1;
END
```

```
1: SYMBOL  (NAME "NUM") (TYPE RECORD) (LIFE STACK) (GLOBAL NO)
           (FIELDS ..., 3:, ...)
3: SYMBOL  (NAME "RI") (TYPE UNION) (FIELDS 4: 5:)
4: SYMBOL  (NAME "R") (TYPE FLOAT) (RANGE 1E0 1E1) (PRECISION 3)
5: SYMBOL  (NAME "I") (TYPE INT) (RANGE 1 10)
6: LITERAL  (VALUE 1) (TYPE INT)

20: TREE  (OP :=) (ANCESTOR 0:) (SUBNODES 21: 26:)

21: TREE  (OP SUBFIELD) (ANCESTOR 20:) (SUBNODES 22: 23:)
22: LEAF  (OP SYMBOL) (ANCESTOR 21:) (VALUE 1:)
23: TREE  (OP DISCRIMINATE) (ANCESTOR 21:) (SUBNODES 24: 25:)
24: LEAF  (OP SYMBOL) (ANCESTOR 23:) (VALUE 3:)
25: LEAF  (OP SYMBOL) (ANCESTOR 23:) (VALUE 5:)

26: TREE  (OP +INT) (ANCESTOR 20:) (SUBNODES 27: 32:)
27: TREE  (OP SUBFIELD) (ANCESTOR 26:) (SUBNODES 28: 29:)
28: LEAF  (OP SYMBOL) (ANCESTOR 27:) (VALUE 1:)
29: TREE  (OP DISCRIMINATE) (ANCESTOR 27:) (SUBNODES 30: 31:)
30: LEAF  (OP SYMBOL) (ANCESTOR 29:) (VALUE 3:)
31: LEAF  (OP SYMBOL) (ANCESTOR 29:) (VALUE 5:)
32: LEAF  (OP LITERAL) (ANCESTOR 26:) (VALUE 6:)
```

Notes: The selection of the variant type within the record is repeated twice (23: - 25: and 29: - 31:). The **subfield** operator then accesses that subfield (the current variant) in the record. Since the choice of the I tag within RI implies that the record field selected is an INT, the + in 26: is coerced to +INT.

In the union RI, only node 3: represents an actual storage location. Nodes 4: and 5: are dummy Symbol nodes whose purpose is to specify the type of the union field when that particular variant is selected.

A.6 Variant records and tags

```
BEGIN
  VAR  u : UNION  [a : INT(1..10), b : BOOL],
       whichType : INT(0..5);

  CASE u.tag OF
    WHEN .a =>  whichType := 1;
    WHEN .b =>  whichType := 2;
    ELSE whichType := 0;
  END CASE

END
```

```
1: SYMBOL  (NAME "U") (TYPE UNION) (LIFE STACK) (GLOBAL NO)
           (FIELDS 2: 3:) (TAG 4:)
2: SYMBOL  (NAME "A") (TYPE INT) (RANGE 1 10)
3: SYMBOL  (NAME "B") (TYPE BOOL)
4: SYMBOL  (TYPE INT) (RANGE 1 2)
5: LITERAL (VALUE TAG) (TYPE BUILTIN)
6: SYMBOL  (NAME "WHICHTYPE") (TYPE INT) (RANGE 0 5)
7: LITERAL (VALUE 0) (TYPE INT)
8: LITERAL (VALUE 1) (TYPE INT)
9: LITERAL (VALUE 2) (TYPE INT)

20: TREE  (OP CASE) (ANCESTOR 0:) (SUBNODES 21: 24: 30: 36:)
21: TREE  (OP INQUIRY) (ANCESTOR 20:) (SUBNODES 22: 23:)
22: LEAF  (OP SYMBOL) (ANCESTOR 21:) (VALUE 1:)
23: LEAF  (OP LITERAL) (ANCESTOR 21:) (VALUE 5:)
24: TREE  (OP ACASE) (ANCESTOR 20:) (SUBNODES 25: 28:)
25: TREE  (OP :=) (ANCESTOR 24:) (SUBNODES 26: 27:)
26: LEAF  (OP SYMBOL) (ANCESTOR 25:) (VALUE 6:)
27: LEAF  (OP LITERAL) (ANCESTOR 25:) (VALUE 8:)
28: TREE  (OP ECOND) (ANCESTOR 24:) (SUBNODES 29:)
29: LEAF  (OP LITERAL) (ANCESTOR 28:) (VALUE 8:)

30: TREE  (OP ACASE) (ANCESTOR 20:) (SUBNODES 31: 34:)
31: TREE  (OP :=) (ANCESTOR 30:) (SUBNODES 32: 33:)
32: LEAF  (OP SYMBOL) (ANCESTOR 31:) (VALUE 6:)
33: LEAF  (OP LITERAL) (ANCESTOR 31:) (VALUE 9:)
34: TREE  (OP ECOND) (ANCESTOR 30:) (SUBNODES 35:)
35: LEAF  (OP LITERAL) (ANCESTOR 34:) (VALUE 9:)
36: TREE  (OP :=) (ANCESTOR 20:) (SUBNODES 37: 38:)
37: LEAF  (OP SYMBOL) (ANCESTOR 36:) (VALUE 6:)
38: LEAF  (OP LITERAL) (ANCESTOR 36:) (VALUE 7:)
```

Notes: The CASE consists of a selection expression (21: - 23:), two "<cond> => <expr>" ACASEs (24: - 29: , 30: - 35:), and an else expression (36: - 38:). Each case has a single guarding condition so each **acase** has first operand the assignment (executed expression) and second operand the guard. The guards are integer constants since they are elements from the enumerated type which identifies the different choices of a variant field within the union. Accordingly, the variable "whichType" is essentially set to "u.tag" . The else clause handles the case where u.tag is undefined. The second **acase** operand was chosen to be **econd** for an expression condition (as the Ada semantics are unclear, it could equally well have been a **rcond** with range of, e.g., 2 to 2).

The TAG attribute for the union u (node 1:) points to a dummy Symbol node (4:) which identifies the current variant. Here this dummy is of type ENUM [.a,.b] which the front end changes into an INT subtype of range (1..2).

The **Inquiry** operator in 21: gives the current value of the TAG of "u" (defined by the type from the last time u was changed).

If the Pascal (or Green) format was used for discriminated unions (i.e. a CASE on the variant choices within a record declaration), there would be an explicit name for the tag. Then there would be a NAME attribute in node 4: and the INQUIRY in nodes 21: - 23: would be replaced by a single Leaf node which referenced the Symbol node 4: for the tag.

No runtime checks are made here to insure that reference to variants conforms to the current tag setting. If such checks are desired, they must be represented explicitly in the tree by constructs which compare the type of the variant to the type of the tag and signal the appropriate exceptional condition when necessary.

A.7 Attribute inquiry using a Case

```
BEGIN
  ...
  BEGIN
    VAR word : STRING (10);
      ...
    CASE word.length OF
      WHEN 1..3 , 5..7  =>  f(0.5);
      WHEN max - 1      =>  f(1.0);
      ELSE   f(0.0)
    END CASE
  END
  ...
END
```

```
1: - 10: are Literal nodes of type INT defining the integers 1-10
11: SYMBOL  (NAME "WORD") (TYPE STRING) (LENGTH 10)
12: SYMBOL  (NAME "F") (TYPE ROUTINE) (LIFE STACK) (GLOBAL NO)
            (EFFECT PURE)
13: LITERAL (VALUE LENGTH) (TYPE BUILTIN)
14: LITERAL (VALUE 0.0) (TYPE FIXED) (DELTA 0.1)
15: LITERAL (VALUE 0.5) (TYPE FIXED) (DELTA 0.1)
16: LITERAL (VALUE 1.0) (TYPE FIXED) (DELTA 0.1)
17: SYMBOL  (NAME "MAX") (TYPE INT) (RANGE 1 10)

20: TREE  (OP CASE) (ANCESTOR 0:) (SUBNODES 21: 24: 34: 42:)
21: TREE  (OP INQUIRY) (ANCESTOR 20:) (SUBNODES 22: 23:)
22: LEAF  (OP SYMBOL) (ANCESTOR 21:) (VALUE 11:)
23: LEAF  (OP LITERAL) (ANCESTOR 21:) (VALUE 13:)

24: TREE  (OP ACASE) (ANCESTOR 20:) (SUBNODES 25: 28: 31:)
25: TREE  (OP CALL) (ANCESTOR 24:) (SUBNODES 26: 27:)
26: LEAF  (OP SYMBOL) (ANCESTOR 25:) (VALUE 12:)
27: LEAF  (OP LITERAL) (ANCESTOR 25:) (VALUE 15:)
28: TREE  (OP RCOND) (ANCESTOR 24:) (SUBNODES 29: 30:)
29: LEAF  (OP LITERAL) (ANCESTOR 28:) (VALUE 1:)
30: LEAF  (OP LITERAL) (ANCESTOR 28:) (VALUE 3:)
31: TREE  (OP RCOND) (ANCESTOR 24:) (SUBNODES 32: 33:)
32: LEAF  (OP LITERAL) (ANCESTOR 31:) (VALUE 5:)
33: LEAF  (OP LITERAL) (ANCESTOR 31:) (VALUE 7:)

34: TREE  (OP ACASE) (ANCESTOR 20:) (SUBNODES 35: 38:)
35: TREE  (OP CALL) (ANCESTOR 34:) (SUBNODES 36: 37:)
36: LEAF  (OP SYMBOL) (ANCESTOR 35:) (VALUE 12:)
37: LEAF  (OP LITERAL) (ANCESTOR 35:) (VALUE 16:)
38: TREE  (OP ECOND) (ANCESTOR 34:) (SUBNODES 39:)
39: TREE  (OP -) (ANCESTOR 38:) (SUBNODES 40: 41:)
40: LEAF  (OP SYMBOL) (ANCESTOR 39:) (VALUE 17:)
41: LEAF  (OP LITERAL) (ANCESTOR 39:) (VALUE 1:)

42: TREE  (OP CALL) (ANCESTOR 20:) (SUBNODES 43: 44:)
43: LEAF  (OP SYMBOL) (ANCESTOR 42:) (VALUE 12:)
44: LEAF  (OP LITERAL) (ANCESOR 42:) (VALUE 14:)
```

Notes: The CASE consists of a selection expression (21: - 23:), two acases (24: - 33:, 34: - 41:) and an else expression (42: - 44:). The selection is an attribute inquiry on the Length attribute of the string variable "word". The first case conditional guard is a list of two ranges (thus rcond is used). The second case guard is an expression involving the variable "max" set somewhere previously (thus econd is used).