An Analyzer for Pascal

W. M. Waite

August 27, 2008

This document describes an analyzer for Pascal (ANSI/IEEE 770X3.97-1983). It was generated from an Eli[1] specification of that analyzer.

An abstract syntax tree structure describing the essential semantics of Pascal is given in Section 1, along with the Eli specifications necessary to build the AST corresponding to any legal program.

The computations specified in Sections 2 and 3 decorate the nodes of the tree with information about the binding of identifiers and the type of expressions, respectively. Section 4 uses these decorations to report semantic errors.

Eli can generate an executable analyzer for Pascal from the specifications used to derive this document. If these specifications are combined with Eli specifications for translation and encoding tasks, Eli can generate a complete compiler for Pascal.

The translation specification will need to deal with the following important issues:

- Overlaid storage
- Insertion of type conversion operators

[1] http://eli-project.sourceforge.net

Contents

Structure.lido[1]:

```
RULE root: Source  ::=    program                                        END;
RULE pgrm: program ::=    'program' identifier PgmPars ';' block '.' END;
RULE pgpl: PgmPars LISTOF PgmPar                                         END;
RULE pgpr: PgmPar  ::=    identifier                                     END;
RULE blck: block   ::=    Decls 'begin' StmtList 'end'                   END;
```

Declarations[2]
Statements[6]
Expressions[8]
Identifiers[11]

This macro is attached to a product file.

Figure 1: The top-level program structure

1 Structural Analysis

The structural analyzer builds an abstract syntax tree (AST) from the text of a source program. Eli generates a structural analyzer from specifications of the

- AST structure (Section 1.1),
- phrase structure (Section 1.2), and
- basic symbols and comments (Section 1.3).

1.1 The Abstract Syntax Tree

A compiler must carry out a number of computations over the abstract syntax tree in the process of analyzing and translating the source program. Simplifying these computations is the primary goal of AST design. While the AST certainly reflects the phrase structure of the source program, it may differ significantly in certain respects.

Attribute grammars describe both the structure of the AST and the computations carried out over it. Figure 1, written in LIDO, is the attribute grammar fragment describing the overall structure of a Pascal program. Each rule describes a node of the AST, and corresponds to a class in an object-oriented implementation. The identifier following the keyword `RULE` names the class of the object that would represent such a node. These are the only classes that can be instantiated. (It is not necessary to name the rules in a LIDO specification, because Eli will generate unique names if none are given. Rules are named in Figure 1 to make it easier to discuss them.)

An identifier that precedes `::=` or `LISTOF` in some rule is called a *nonterminal*; all other identifiers are *terminals*. Nonterminals following `::=` represent

subtrees, while terminals represent values that are not subtrees. (For example, `identifier` is a terminal representing an identifier appearing in the source program.)

Terminals can also be thought of as class names, but these classes are defined outside of the LIDO specification. They do not represent tree nodes, but rather values that are components of a rule's object. Objects of class `pgpr` therefore have no children, but each stores a representation of an identifier appearing in the source program.

A nonterminal (such as `block` in Figure 1) names the abstract class that characterizes the contexts in which the construct can appear. Each rule class (such as `body` in Figure 1) is a subclass of the abstract class named by the nonterminal preceding `::=` or `LISTOF`.

Each rule containing `::=` describes a node with a fixed number of children and/or component values. Nonterminals following the `::=` specify children, in left-to-right order. Each child must be represented by an object belonging to a subclass of the abstract class named by the nonterminal.

Each rule containing `LISTOF` describes a node with an arbitrary number of children (including none at all). Nonterminals following `LISTOF` (separated by vertical bars) specify the possible children. There may be any number of children corresponding to each nonterminal.

Literals in Figure 1 do not represent information present in the abstract syntax tree; they are used solely to establish a correspondence between the abstract syntax tree nodes and the phrase structure of the input.

1.1.1 Declarations

The declaration part of a Pascal `block` (Figure 2) defines labels, constants, types, variables, and routines (procedures or functions) respectively. Individual declarations must occur in the order given in the previous sentence. As far as the semantics of the language are concerned, however, that order is actually immaterial. Because the attribute grammar is concerned with semantics, it simply states that the declaration part of a block contains some number of declarations of various kinds.

Pascal constants (Figure 3) may specify (signed) numbers, characters, or strings. Constants may also be named, and the names used in place of the value. There is no syntactic difference between a character constant and a string constant; if the constant specifies a single character then its type is `char`, otherwise it has a string type. By introducing the `litr` rule, we provide a place to to check the length of the character string and establish the appropriate type.

Types (Figure 4) can be represented by identifiers or by constructors describing the structure of the type. Most constructors have very similar semantics, and can therefore be represented by a single nonterminal (`TypeDenoter`). Record types have a complex inner structure that requires additional information. Therefore it is convenient to use a second nonterminal (`Record`) at the root of that definition. Note that rule `fldl` provides an example of a node with an arbitrary collection of children of different kinds. Actually, the record will

Declarations[2]:

```
RULE dcls: Decls    LISTOF Decl                          END;
RULE dlbl: Decl     ::=    LblIdDef                      END;
RULE dcon: Decl     ::=    ConIdDef '=' constant         END;
RULE dtyp: Decl     ::=    TypIdDef '=' type             END;
RULE dvar: Decl     ::=    VrblIds ':' type              END;
RULE dprc: Decl     ::=    'procedure' PrcIdDef ProcBody END;
RULE dfnc: Decl     ::=    'function'  FncIdDef FuncBody END;
RULE dfwd: Decl     ::=    'function'  FncIdUse ';' Body END;
RULE body: Body     ::=    block                         END;
RULE lbld: LblIdDef ::=    integer_constant             END;
RULE cnsd: ConIdDef ::=    identifier                    END;
RULE typd: TypIdDef ::=    identifier                    END;
RULE vidl: VrblIds  LISTOF VblIdDef                      END;
RULE vbld: VblIdDef ::=    identifier                    END;
RULE prcd: PrcIdDef ::=    identifier                    END;
RULE fncd: FncIdDef ::=    identifier                    END;
```

This macro is defined in definitions 2, 3, 4, and 5.
This macro is invoked in definition 1.

Figure 2: The structure of a declaration part

Declarations[3]:

```
RULE sgni: constant  ::= csign integer_constant END;
RULE sgnr: constant  ::= csign real_constant    END;
RULE sidn: constant  ::= csign ConIdUse         END;
RULE cint: constant  ::= integer_constant       END;
RULE cflt: constant  ::= real_constant          END;
RULE cidn: constant  ::= ConIdUse               END;
RULE cstr: constant  ::= Literal                END;
RULE cpls: csign     ::= '+'                    END;
RULE cmin: csign     ::= '-'                    END;
RULE cnsu: ConIdUse  ::= identifier             END;
RULE litr: Literal   ::= character_string       END;
```

This macro is defined in definitions 2, 3, 4, and 5.
This macro is invoked in definition 1.

Figure 3: The structure of a constant

Declarations[4]:

```
RULE pack: type        ::=    'packed' type                  END;
RULE tyid: type        ::=    TypIdUse                       END;
RULE tden: type        ::=    TypeDenoter                    END;
RULE rdec: type        ::=    Record                         END;
RULE typu: TypIdUse    ::=    identifier                     END;
RULE enum: TypeDenoter ::=    '(' Enumerate ')'              END;
RULE rnge: TypeDenoter ::=    constant '..' constant         END;
RULE arry: TypeDenoter ::=    'array' '[' type ']' 'of' type END;
RULE sett: TypeDenoter ::=    'set'  'of' type               END;
RULE file: TypeDenoter ::=    'file' 'of' type               END;
RULE pntr: TypeDenoter ::=    '^' type                       END;
RULE encl: Enumerate   LISTOF ConIdDef                       END;
RULE recd: Record      ::=    'record' Fields 'end'          END;
RULE fldl: Fields      LISTOF Decl | var_part                END;
RULE vprt: var_part    ::=    'case' var_sel 'of' Variants   END;
RULE vtag: var_sel     ::=    TagIdDef ':' TypIdUse          END;
RULE vunt: var_sel     ::=               TypIdUse            END;
RULE vars: Variants    LISTOF Variant                        END;
RULE vrnt: Variant     ::=    constants ':' '(' Fields ')'   END;
RULE vsel: constants   LISTOF constant                       END;
RULE tagd: TagIdDef    ::=    identifier                     END;
```

This macro is defined in definitions 2, 3, 4, and 5.
This macro is invoked in definition 1.

Figure 4: The structure of a type definition

Declarations[5]:

```
RULE pbdy: ProcBody  ::=    Formals ';' block              END;
RULE fbdy: FuncBody  ::=    Formals ':' TypIdUse ';' block END;
RULE fnof: FuncBody  ::=            ':' TypIdUse ';' block END;
RULE form: Formals   LISTOF Formal                         END;
RULE fval: Formal    ::=    FrmlIds ':' type               END;
RULE fvar: Formal    ::=    'var' FrmlIds ':' type         END;
RULE fprc: Formal    ::=    'procedure' FmlIdDef ProcHead  END;
RULE ffnc: Formal    ::=    'function'  FmlIdDef FuncHead  END;
RULE prhd: ProcHead  ::=    Formals                        END;
RULE fnhd: FuncHead  ::=    Formals ':' TypIdUse           END;
RULE fhnf: FuncHead  ::=            ':' TypIdUse           END;
RULE fidl: FrmlIds   LISTOF FmlIdDef                       END;
RULE bfwd: block     ::=    'forward'                      END;
RULE bext: block     ::=    'external'                     END;
```

This macro is defined in definitions 2, 3, 4, and 5.
This macro is invoked in definition 1.

Figure 5: The structure of a routine declaration

have no more than one `var_part`, but there is no need to carry that information into the AST explicitly.

A routine declaration (Figure 5 may use the directive `forward` or `external` in lieu of a block. Although these directives are defined as possible descendants of `block`, the phrase structure described later restricts then to blocks defining routines.

1.1.2 Statements

Statements[6]:

```
RULE stms: StmtList  LISTOF statement                                  END;
RULE locd: statement ::=    LblIdUse ':' statement                     END;
RULE emty: statement ::=                                               END;
RULE assn: statement ::=    variable ':=' expression                   END;
RULE call: statement ::=    ProcCall                                   END;
RULE goto: statement ::=    'goto' LblIdUse                            END;
RULE cmpd: statement ::=    'begin' StmtList 'end'                     END;
RULE ones: statement ::=    'if' expression 'then' statement           END;
RULE twos: statement ::=    'if' expression 'then' statement
                                            'else' statement           END;
RULE cstm: statement ::=    'case' expression 'of' cases 'end'         END;
RULE rpts: statement ::=    'repeat' StmtList 'until' expression       END;
RULE whil: statement ::=    'while' expression 'do' statement          END;
```

```
RULE foru: statement ::=    'for' ExpIdUse ':=' expression
                              'to' expression 'do' statement          END;
RULE ford: statement ::=    'for' ExpIdUse ':=' expression
                              'downto' expression 'do' statement      END;
RULE with: statement ::=    'with' WithVar 'do' WithBody              END;
RULE casl: cases     LISTOF case                                      END;
RULE celt: case      ::=    selectors ':' statement                   END;
RULE csel: selectors LISTOF constant                                  END;
RULE wcls: WithVar   ::=    variable                                  END;
RULE wbdy: WithBody  ::=    statement                                 END;
RULE iost: statement ::=    InOutStmt                                 END;
```

This macro is defined in definitions 6 and 7.
This macro is invoked in definition 1.

Statements[7]:

```
RULE proc: ProcCall  ::=    PrcIdUse PrcArgs                          END;
RULE parg: PrcArgs   LISTOF Actual                                    END;
RULE arge: Actual    ::=    expression                                END;
RULE read: InOutStmt ::=    'read'    '(' RdArgs ')'                  END;
RULE rlin: InOutStmt ::=    'readln'                                  END;
RULE rdln: InOutStmt ::=    'readln'  '(' RdArgs ')'                  END;
RULE writ: InOutStmt ::=    'write'   '(' WrtArgs ')'                 END;
RULE wlin: InOutStmt ::=    'writeln'                                 END;
RULE wrln: InOutStmt ::=    'writeln' '(' WrtArgs ')'                 END;
RULE rarg: RdArgs    LISTOF RdArg                                     END;
RULE rrgv: RdArg     ::=    variable                                  END;
RULE warg: WrtArgs   LISTOF WrtArg                                    END;
RULE wrgr: WrtArg    ::=    expression ':' expression ':' expression  END;
RULE wrgw: WrtArg    ::=    expression ':' expression                 END;
RULE wrge: WrtArg    ::=    expression                                END;
```

This macro is defined in definitions 6 and 7.
This macro is invoked in definition 1.

1.1.3 Expressions

Expressions[8]:

```
RULE vbid: variable  ::=    ExpIdUse                                  END;
RULE indx: variable  ::=    variable '[' Subscript ']'                END;
RULE fldv: variable  ::=    variable '.' FldIdUse                     END;
RULE dref: variable  ::=    variable '^'                              END;
RULE subs: Subscript ::=    expression                                END;
```

This macro is defined in definitions 8, 9, and 10.
This macro is invoked in definition 1.

Expressions[9]:

```
RULE iexp: expression ::=    integer_constant                        END;
RULE rexp: expression ::=    real_constant                           END;
RULE sexp: expression ::=    Literal                                  END;
RULE nexp: expression ::=    'nil'                                    END;
RULE vexp: expression ::=    variable                                 END;
RULE func: expression ::=    FncIdUse FncArgs                         END;
RULE setx: expression ::=    '[' Members ']'                          END;
RULE eset: expression ::=    '['         ']'                          END;
RULE dyad: expression ::=    expression operator expression           END;
RULE mnad: expression ::=               operator expression           END;
RULE farg: FncArgs    LISTOF Actual                                   END;
RULE meml: Members    LISTOF Member                                   END;
RULE meme: Member     ::=    expression                               END;
RULE memr: Member     ::=    expression '..' expression               END;
```

This macro is defined in definitions 8, 9, and 10.
This macro is invoked in definition 1.

Expressions[10]:

```
RULE equl: operator ::= '='   END;
RULE neql: operator ::= '<>'  END;
RULE less: operator ::= '<'   END;
RULE grtr: operator ::= '>'   END;
RULE lseq: operator ::= '<='  END;
RULE greq: operator ::= '>='  END;
RULE memb: operator ::= 'in'  END;
RULE plus: operator ::= '+'   END;
RULE mins: operator ::= '-'   END;
RULE disj: operator ::= 'or'  END;
RULE star: operator ::= '*'   END;
RULE slsh: operator ::= '/'   END;
RULE idiv: operator ::= 'div' END;
RULE remr: operator ::= 'mod' END;
RULE conj: operator ::= 'and' END;
RULE linv: operator ::= 'not' END;
```

This macro is defined in definitions 8, 9, and 10.
This macro is invoked in definition 1.

1.1.4 Identifiers

Identifiers[11]:

```
RULE fmld: FmlIdDef ::= identifier       END;
```

```
RULE lblu: LblIdUse ::= integer_constant END;
RULE prcu: PrcIdUse ::= identifier       END;
RULE fncu: FncIdUse ::= identifier       END;
RULE fldu: FldIdUse ::= identifier       END;
RULE exid: ExpIdUse ::= identifier       END;
```

This macro is invoked in definition 1.

1.2 Phrase structure

Most of the phrase structure of Pascal can be deduced from the abstract syntax given in Section 1.1. In a number of cases, however, the abstract syntax is either ambiguous or designed to capture the semantics of a construct more cleanly than the phrase structure allows. *Concrete syntax* rules are introduced to define the phrase structure in those situations.

Structure.con[12]:

This macro is attached to a product file.

Eli uses pattern matching to determine the relationship between phrases defined by a concrete syntax (Section 1.2) and AST nodes defined by the abstract syntax (Section 1.1). It recognizes simple identities like that of rule `pgrm` in Figure 1. `LISTOF` rules are properly associated with concrete syntax rules that use extended BNF notation to describe iteration.

The concrete syntax for Pascal uses a number of symbols that do not appear in the abstract grammar. For example, the operator precedence rules of the language are described by using distinct symbols for expressions at each precedence level. *Symbol mappings* convert all of these distinct expression symbols into the single symbol `expression` used in the abstract grammar.

Several shorthand notations, such as multi-dimensional arrays, are incorporated into the concrete syntax. These shorthands are converted into their long form in the abstract syntax by *rule mappings*.

Eli cannot deduce these mappings from the grammars themselves; an additional specification is needed:

Structure.map[13]:

```
MAPSYM

expression ::= simple_expression term factor .
constant   ::= selector .

operator ::= sign multiplying_operator adding_operator relational_operator .

Decl ::=
  constant_definition
  type_definition record_section
  variable_declaration
  procedure_declaration
  function_declaration .

statement ::= unlabeled_statement .
```

List symbol mapping[16]
Extension symbol mapping[23]

```
MAPRULE
```

Extension rule mapping[24]

This macro is attached to a product file.

1.2.1 Constants

Constants[14]:

```
constant: [csign] (integer_constant / real_constant / ConIdUse) / Literal .
selector: [csign] (integer_constant / real_constant / ConIdUse) / Literal .
```

This macro is invoked in definition 12.

1.2.2 Lists

Lists[15]:

```
PgmPars: [ '(' (PgmPar // ',') ')' ] .
Enumerate: ConIdDef // ',' .
constants: constant // ',' .
selectors: selector // ',' .
Variants:  variant_list .
variant_list: / Variant / Variant ';' variant_list .
VrblIds:   VblIdDef // ',' .
FrmlIds:   FmlIdDef // ',' .
Formals:   '(' (Formal // ';') ')'   .
FncArgs:   '(' (Actual // ',') ')'   .
```

```
PrcArgs: [ '(' (Actual // ',') ')' ] .
RdArgs:         RdArg  // ','        .
WrtArgs:        WrtArg // ','        .
Members:        Member // ','        .
StmtList: statement // ';' .
cases: case_list .
case_list: / case / case ';' case_list .

ProcBody:   optFormals ';' block .
ProcHead:   optFormals .
optFormals: [ '(' (Formal // ';') ')' ] .
```

This macro is defined in definitions 15.
This macro is invoked in definition 12.

List symbol mapping[16]:

```
Formals ::= optFormals .
```

This macro is invoked in definition 13.

1.2.3 Fields

Fields[17]:

```
Fields: [ fixed_part [';' [var_part]] / var_part ] .
fixed_part: record_section / fixed_part ';' record_section .
record_section: VrblIds ':' type .
```

This macro is invoked in definition 12.

1.2.4 Statement labeling

Statement labeling[18]:

```
statement: [ LblIdUse ':' ] unlabeled_statement .
unlabeled_statement: .
unlabeled_statement: variable ':=' expression .
unlabeled_statement: ProcCall .
unlabeled_statement: 'goto' LblIdUse .
unlabeled_statement: 'begin' StmtList 'end' .
unlabeled_statement: 'case' expression 'of' cases 'end' .
unlabeled_statement: 'repeat' StmtList 'until' expression .
unlabeled_statement: 'while' expression 'do' statement .
unlabeled_statement: 'for' ExpIdUse ':=' expression
                              'to' expression 'do' statement .
unlabeled_statement: 'for' ExpIdUse ':=' expression
                              'downto' expression 'do' statement .
unlabeled_statement: InOutStmt .
```

This macro is invoked in definition 12.

1.2.5 Precedence and association

Precedence and association[19]:

```
expression: simple_expression [ relational_operator simple_expression ] .

simple_expression: [sign] term / simple_expression adding_operator term .

sign: '+' / '-' .
csign: '+' / '-' .

term: factor / term multiplying_operator factor .

factor:
  '(' expression ')' /
  'not' factor /
  FncIdUse FncArgs /
  '[' Members ']' /
  '['         ']' /
  variable /
  integer_constant /
  real_constant /
  Literal /
  'nil' .

multiplying_operator: '*' / '/' / 'div' / 'mod' / 'and' .

adding_operator: '+' / '-' / 'or' .

relational_operator: '=' / '<>' / '<' / '>' / '<=' / '>=' / 'in' .

variable: ExpIdUse .
```

This macro is defined in definitions 19.
This macro is invoked in definition 12.

1.2.6 Dangling else

Dangling else[20]:

```
unlabeled_statement:
  'if' expression 'then' statement $'else' /
  'if' expression 'then' statement  'else' statement .
```

This macro is invoked in definition 12.

1.2.7 Declaration sequence

Declaration sequence[21]:

```
Decls:
  [ 'label' (LblIdDef // ',') ';' ]
  [ 'const' (constant_definition ';')* ]
  [ 'type'  (type_definition ';')* ]
  [ 'var'   (variable_declaration ';')* ]
  (procedure_declaration ';' / function_declaration ';')* .

constant_definition:   ConIdDef '=' constant .
type_definition:       TypIdDef '=' type .
variable_declaration:  VrblIds ':' type .
procedure_declaration: 'procedure' PrcIdDef ProcBody.
function_declaration:
  'function'  FncIdDef FuncBody /
  'function'  FncIdUse ';' Body .
```

This macro is defined in definitions 21.
This macro is invoked in definition 12.

1.2.8 Extensions

Extensions[22]:

```
TypeDenoter:
  'array' '[' type array_tail /
  'array' '[' type ']' 'of' type .
array_tail:
  ',' type array_tail /
  ',' type ']' 'of' type .

unlabeled_statement: 'with' variable with_tail .
with_tail:
  ',' variable with_tail /
  'do' statement .

variable: subscript_head ']' .
subscript_head:
  subscript_head ',' Subscript /
  variable '[' Subscript .
```

This macro is invoked in definition 12.

Extension symbol mapping[23]:

```
TypeDenoter ::= array_tail .
statement ::= with_tail .
variable ::= subscript_head .
```

This macro is invoked in definition 13.

Structure.gla[25]:

```
identifier:             PASCAL_IDENTIFIER
integer_constant:       PASCAL_INTEGER          [mkidn]
real_constant:          PASCAL_REAL             [mkidn]
character_string:       PASCAL_STRING           [mkidn]
                        PASCAL_COMMENT
```

This macro is attached to a product file.

Figure 6: Non-literal Basic Symbols and Coments

Extension rule mapping[24]:

```
TypeDenoter: 'array' '[' type array_tail
  < 'array' '[' $1 ']' 'of' $2 > .
array_tail: ',' type array_tail
  < 'array' '[' $1 ']' 'of' $2 > .
array_tail: ',' type ']' 'of' type
  < 'array' '[' $1 ']' 'of' $2 > .

unlabeled_statement: 'with' variable with_tail
  < 'with' $1 'do' $2 > .
with_tail: ',' variable with_tail
  < 'with' $1 'do' $2 > .

subscript_head: variable '[' Subscript
  < $1 '[' $2 ']' > .
subscript_head: subscript_head ',' Subscript
  < $1 '[' $2 ']' > .
```

This macro is invoked in definition 13.

1.3 Basic symbols and comments

The generated structural analyzer must read a sequence of characters from a file and build the corresponding abstract syntax tree. Eli can extract the definitions of literal basic symbols from the grammars, but neither Section 1.2 nor Figure 1 describes the sequence of characters making up identifiers and numbers. There is also no indication of what constitutes a comment.

Figure 6 contains the necessary definitions. The Eli library provides *canned descriptions* (such as `PASCAL_IDENTIFIER`) for all of these character sequences. (White space will be automatically ignored by an Eli-generated scanner unless the user explicitly defines some other behavior.)

The canned descriptions of `PASCAL_INTEGER` and `PASCAL_REAL` do not guarantee that each distinct denotation has a unique internal representation no matter how many times it appears in the source text. It is convenient to have unique

representations for denotations, and hence Figure 6 overrides the creation of their internal representations. The token processor `mkidn` enters the scanned character sequence into the string table if and only if it is not already in that table. It then uses the index of the (newly or previously) stored string as the internal representation of the basic symbol.

Keywords.gla[26]:

```
$[a-z]+
```

This macro is attached to a non-product file.

Structure.specs[27]:

```
Keywords.gla :kwd
```

This macro is attached to a product file.

2 Name Analysis

This specification binds all unqualified identifiers according to the scope rules of Pascal, defined in Section 6.2.2 of ANSI/IEEE 770X3.97-1983. Bindings for qualified identifiers and field identifiers within a `with` statement depend on the type analysis defined in Section 3. Violations of context conditions involving identifier definition are reported by the specification of Section 4.

Pascal has a single name space. Every identifier or label has a defining point within a region. The scope of the defining point is the entire region, with the exception of nested regions containing a defining point for the same identifier or label. Eli's `AlgScope` module implements this functionality; other modules will be instantiated later in this section as needed:

Name.specs[1]:

```
$/Name/AlgScope.gnrc :inst
```
Instantiate appropriate modules[9]

This macro is attached to a product file.

A precondition for using `AlgScope` is that every identifier occurrence have a `Sym` attribute:

Name.lido[2]:

```
CLASS SYMBOL IdentOcc COMPUTE SYNT.Sym=TERM; END;
```

Regions[3]
Defining occurrences[11]
Applied occurrences[12]

This macro is attached to a product file.

2.1 Regions

The abstract syntax tree combines formal parameter lists and routine bodies into single trees that do not contain the routine identifier. These trees form the regions within which identifiers are declared:

Regions[3]:

```
SYMBOL program  INHERITS RangeScope END;
SYMBOL ProcBody INHERITS RangeScope END;
SYMBOL FuncBody INHERITS RangeScope END;
SYMBOL     Body INHERITS RangeScope END;
SYMBOL ProcHead INHERITS RangeScope END;
SYMBOL FuncHead INHERITS RangeScope END;
```

This macro is defined in definitions 3, 4, 5, 7, and 8.
This macro is invoked in definition 2.

If a function or procedure is declared `forward`, then the `Formals` at the forward declaration combine with the `block` at the required subsequent identification. We use the attribute `IsForward` to signal the presence of a forward declaration. It is set to 0 by a symbol computation that is overridden in exactly the case of the forward declaration:

Regions[4]:

```
ATTR IsForward: int;

SYMBOL block                COMPUTE  SYNT.IsForward=0; END;
RULE:  block ::= 'forward' COMPUTE block.IsForward=1; END;
```

This macro is defined in definitions 3, 4, 5, 7, and 8.
This macro is invoked in definition 2.

A void chain can be used to ensure that forward declarations are processed in textual order:

Regions[5]:

```
CHAIN ForwardRoutine: VOID;

SYMBOL program COMPUTE CHAINSTART HEAD.ForwardRoutine="yes"; END;
```

This macro is defined in definitions 3, 4, 5, 7, and 8.
This macro is invoked in definition 2.

The environment of a forward-declared procedure or function must be conveyed from the forward declaration to the *identification* (non-forward declaration), and that requires a property of the identifier. Each procedure or function identifier must also carry a property that reflects the state of the forward definition process:

Name.pdl[6]:

```
RtnEnv:   Environment;  "envmod.h"
FwdState: int;
```

This macro is attached to a product file.

`FwdState` is coded as follows:

0 No declaration of this identifer has been seen.

1 A `forward` has been seen, but there has been no identification.

2 This identifier has been declared.

The value of `FwdState` when `ForwardRoutine` reaches a node is coded in the `.FwdState` rule attribute, and is updated as follows:

Regions[7]:

```
ATTR FwdState: int;

RULE: Decl ::= 'procedure' PrcIdDef ProcBody COMPUTE
  .FwdState=GetFwdState(PrcIdDef.Key,0) <- Decl.ForwardRoutine;
  Decl.ForwardRoutine=
    IF(NOT(ProcBody CONSTITUENT block.IsForward SHIELD (Formals,block)),
      ResetFwdState(PrcIdDef.Key,2),
    IF(EQ(.FwdState,0),
      ORDER(
        ResetFwdState(PrcIdDef.Key,1),
        ResetRtnEnv(PrcIdDef.Key,ProcBody.Env))));
END;

RULE: Decl ::= 'function' FncIdDef FuncBody COMPUTE
  .FwdState=GetFwdState(FncIdDef.Key,0) <- Decl.ForwardRoutine;
  Decl.ForwardRoutine=
    IF(NOT(FuncBody CONSTITUENT block.IsForward SHIELD (Formals,block)),
      ResetFwdState(FncIdDef.Key,2),
    IF(EQ(.FwdState,0),
      ORDER(
        ResetFwdState(FncIdDef.Key,1),
        ResetRtnEnv(FncIdDef.Key,FuncBody.Env))));
END;

RULE: Decl ::= 'function' FncIdUse ';' Body COMPUTE
  .FwdState=GetFwdState(FncIdUse.Key,0) <- Decl.ForwardRoutine;
  Decl.ForwardRoutine=ResetFwdState(FncIdUse.Key,2);
END;
```

This macro is defined in definitions 3, 4, 5, 7, and 8.
This macro is invoked in definition 2.

The environment for a procedure or function needs to be set to the saved value under appropriate circumstances:

Regions[8]:

```
RULE: Decl ::= 'procedure' PrcIdDef ProcBody COMPUTE
  ProcBody.Env=
    IF(EQ(.FwdState,1),
      GetRtnEnv(PrcIdDef.Key,NoEnv),
      NewScope(INCLUDING AnyScope.Env));
END;

RULE: Decl ::= 'function' FncIdUse ';' Body COMPUTE
```

```
  Body.Env=
    IF(EQ(.FwdState,1),
      GetRtnEnv(FncIdUse.Key,NoEnv),
      NewScope(INCLUDING AnyScope.Env));
END;
```

This macro is defined in definitions 3, 4, 5, 7, and 8.
This macro is invoked in definition 2.

2.2 Identifiers denoting required entities

Identifiers that denote required constants, types, procedures and functions are used as though their defining points have a region enclosing the program. Eli's pre-defined identifier module implements this functionality.

Instantiate appropriate modules[9]:

```
$/Name/PreDefine.gnrc +referto=identifier :inst
$/Name/PreDefId.gnrc +referto=(Required.d) :inst
```

This macro is defined in definitions 9.
This macro is invoked in definition 1.

All of the required identifiers listed in Appendix C of ANSI/IEEE 770X3.97-1983 are specified in file `Required.d` (Figure 7).

2.3 Defining occurrences

This specification distinguishes defining occurrences syntactically, so that the necessary computations can be inherited from the library:

Defining occurrences[11]:

```
TREE SYMBOL LblIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL ConIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL TypIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL TagIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL VblIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL FmlIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL PrcIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL FncIdDef INHERITS IdentOcc, IdDefScope END;
```

This macro is defined in definitions 11.
This macro is invoked in definition 2.

Required.d[10]:

```
PreDefKey("abs", absKey)
PreDefKey("arctan", arctanKey)
PreDefKey("Boolean", boolKey)
PreDefKey("char", charKey)
PreDefKey("chr", chrKey)
PreDefKey("cos", cosKey)
PreDefKey("dispose", disposeKey)
PreDefKey("eof", eofKey)
PreDefKey("eoln", eolnKey)
PreDefKey("exp", expKey)
PreDefKey("false", falseKey)
PreDefKey("get", getKey)
PreDefKey("input", inputKey)
PreDefKey("integer", intKey)
PreDefKey("ln", lnKey)
PreDefKey("maxint", maxintKey)
PreDefKey("new", newKey)
PreDefKey("odd", oddKey)
PreDefKey("ord", ordKey)
PreDefKey("output", outputKey)
PreDefKey("pack", packKey)
PreDefKey("page", pageKey)
PreDefKey("pred", predKey)
PreDefKey("put", putKey)
PreDefKey("read", readKey)
PreDefKey("readln", readlnKey)
PreDefKey("real", realKey)
PreDefKey("reset", resetKey)
PreDefKey("rewrite", rewriteKey)
PreDefKey("round", roundKey)
PreDefKey("sin", sinKey)
PreDefKey("sqr", sqrKey)
PreDefKey("sqrt", sqrtKey)
PreDefKey("succ", succKey)
PreDefKey("text", textKey)
PreDefKey("true", trueKey)
PreDefKey("trunc", truncKey)
PreDefKey("unpack", unpackKey)
PreDefKey("write", writeKey)
PreDefKey("writeln", writelnKey)
```

This macro is attached to a product file.

Figure 7: Required identifiers

2.4 Applied occurrences

This specification distinguishes applied occurrences syntactically, so that the necessary computations can be inherited from the library:

Applied occurrences[12]:

```
TREE SYMBOL LblIdUse INHERITS IdentOcc, IdUseEnv END;
TREE SYMBOL ConIdUse INHERITS IdentOcc, IdUseEnv END;
TREE SYMBOL TypIdUse INHERITS IdentOcc, IdUseEnv END;
TREE SYMBOL PrcIdUse INHERITS IdentOcc, IdUseEnv END;
TREE SYMBOL FncIdUse INHERITS IdentOcc, IdUseEnv END;
TREE SYMBOL ExpIdUse INHERITS IdentOcc, IdUseEnv END;
```

This macro is defined in definitions 12.
This macro is invoked in definition 2.

3 Type Analysis

This specification implements the Pascal type model, the declaration and use of identifiers representing types and typed entities, and the type analysis of expressions. Those properties of Pascal are defined Sections 6.4-6.7 of ANSI/IEEE 770X3.97-1983.

LIDO computations are used to establish the meanings of identifiers and to analyze the types of operands in expressions.

Type.lido[1]:

```
ATTR Type: DefTableKey;
```

Establish a user-defined type[4]
Packing[5]
Sets of required ordinal types[8]
Constant definitions[25]
Type definitions[27]
Variable declarations[28]
Procedure and function declarations[29]
Expressions[36]
Statements[37]
Qualified identifiers[40]

This macro is attached to a product file.

All of the Eli type analysis modules are required.

Type.specs[2]:

```
$/Type/Typing.gnrc :inst
$/Type/Expression.gnrc :inst
$/Type/PreDefOp.gnrc +referto=(Operator.d) :inst
$/Type/StructEquiv.fw
```
Instantiate appropriate modules[41]

This macro is attached to a product file.

3.1 The Pascal type model

A type model consists of a number of language-defined types and operators, plus facilities for constructing user-defined types. The model is defined primarily with OIL, but this section also contains some LIDO computations.

Type.oil[3]:

Required simple types[6]
Enumerated types[9]
Subrange types[12]

Array types[14]
Set types[17]
File types[20]
Pointer types[22]

This macro is attached to a product file.

Since all of the user-defined types create operators, it's useful to bundle the type denotation and operator definition roles into a single role:

Establish a user-defined type[4]:

```
SYMBOL TypeDenoter INHERITS TypeDenotation, OperatorDefs END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

FIXME: This implementation makes no distinction between packed and unpacked types.

Packing[5]:

```
RULE: type ::= 'packed' type COMPUTE
  type[1].Type=type[2].Type;
END;
```

This macro is invoked in definition 1.

3.1.1 Required simple types

Required simple types[6]:

```
SET ordinalType = [intType, boolType, charType];
SET arithType = [intType, realType];
SET simpleType = [intType, realType, boolType, charType];

COERCION
  (intType): realType;

OPER
  cmpeq, cmpne, cmpls, cmpgt, cmple, cmpge(simpleType,simpleType): boolType;
  pos, neg(arithType): arithType;
  add, sub, mul(arithType,arithType): arithType;
  divi, rem(intType,intType): intType;
  divr(realType,realType): realType;

  inv(boolType): boolType;
  disj, conj(boolType,boolType): boolType;
```

```
  rewriteOp, putOp, resetOp, getOp(textType): voidType;
  readOp(textType, simpleType): voidType;
  rdtextOp(simpleType): voidType;
  readlnOp(textType): voidType;
  wtextOp(writableType): voidType;
  wlistOp(writableType,writableType): voidType;
  wfileOp(textType, writableType): voidType;
  writelnOp(textType): voidType;

  absOp, sqrOp(arithType): arithType;
  sinOp, cosOp, expOp, lnOp, sqrtOp, arctanOp(realType): realType;

  truncOp, roundOp(realType): intType;
  ordOp(ordinalType): intType;
  chrOp(intType): charType;
  succOp, predOp(ordinalType): ordinalType;
  oddOp(intType): boolType;
  txtTest(textType): boolType;
  txtDeref(textType): charType;

COERCION
  (intType): writableType;
  (realType): writableType;
  (boolType): writableType;
  (charType): writableType;
  (stringType): writableType;
  (eofType): boolType;
  (eolnType): boolType;

INDICATION
  equal:        cmpeq;
  lsgt:         cmpne;
  less:         cmpls;
  greater:      cmpgt;
  lessequal:    cmple;
  greaterequal: cmpge;
  plus:         pos, add;
  minus:        neg, sub;
  or:           disj;
  star:         mul;
  slash:        divr;
  div:          divi;
  mod:          rem;
  and:          conj;
  not:          inv;
```

```
rewriteType:  rewriteOp;
putType:      putOp;
resetType:    resetOp;
getType:      getOp;
readType:     readOp, rdtextOp;
readlnType:   readlnOp;
writeType:    wlistOp, wtextOp, wfileOp;
writelnType:  writelnOp, wlistOp, wtextOp, wfileOp;

absType:      absOp;
sqrType:      sqrOp;
sinType:      sinOp;
cosType:      cosOp;
expType:      expOp;
lnType:       lnOp;
sqrtType:     sqrtOp;
arctanType:   arctanOp;

truncType:    truncOp;
roundType:    roundOp;
ordType:      ordOp;
chrType:      chrOp;
succType:     succOp;
predType:     predOp;
oddType:      oddOp;
eofType:      txtTest;
eolnType:     txtTest;

deref:        txtDeref;
```

This macro is invoked in definition 3.

Type.pdl[7]:

```
intKey   -> Defer={intType};
realKey  -> Defer={realType};
boolKey  -> Defer={boolType};
charKey  -> Defer={charType};
textKey  -> Defer={textType};

trueKey  -> TypeOf={boolType};
falseKey -> TypeOf={boolType};
```

Property definitions[11]

This macro is attached to a product file.

The required ordinal types all have corresponding set types:

Sets of required ordinal types[8]:

```
TREE SYMBOL program COMPUTE
  SYNT.GotType=
    ORDER(
      AddTypeToBlock(
        intsetType,
        SetTypes,
        SingleDefTableKeyList(intType)),
      ResetCanonicalSet(intType,intsetType),
      AddTypeToBlock(
        boolsetType,
        SetTypes,
        SingleDefTableKeyList(boolType)),
      ResetCanonicalSet(boolType,boolsetType),
      AddTypeToBlock(
        charsetType,
        SetTypes,
        SingleDefTableKeyList(charType)),
      ResetCanonicalSet(charType,charsetType));
  SYNT.GotOper=
    ORDER(
      InstClass1(setType,FinalType(intsetType),intType),
      MonadicOperator(
        makeset,
        NoOprName,
        intType,
        FinalType(intsetType)),
      InstClass1(setType,FinalType(boolsetType),boolType),
      MonadicOperator(
        makeset,
        NoOprName,
        boolType,
        FinalType(boolsetType)),
      InstClass1(setType,FinalType(charsetType),charType),
      MonadicOperator(
        makeset,
        NoOprName,
        charType,
        FinalType(charsetType)));
END;

RULE: Source ::= program COMPUTE
  Source.GotType=program.GotType;
  Source.GotOper=program.GotOper;
END;
```

This macro is invoked in definition 1.

3.1.2 Enumerated types

Enumerated types[9]:

```
CLASS enumType() BEGIN
  OPER
    enumOrd(enumType): intType;
    enumeq, enumne, enumls, enumgt, enumle,
      enumge(enumType,enumType): boolType;
END;

INDICATION
  ordType: enumOrd;
  equal: enumeq;
  lsgt: enumne;
  less: enumls;
  greater: enumle;
  lessequal: enumle;
  greaterequal: enumge;
```

This macro is invoked in definition 3.

Each enumerated type needs a set type, because set expressions can be used as subexpressions without the corresponding set type being declared. The reason for this *canonical set* type is that in a set expression made up of constants the constants are of the base type.

Establish a user-defined type[10]:

```
ATTR CanonicalSet: DefTableKey;

RULE: TypeDenoter ::=  '(' Enumerate ')' COMPUTE
  .CanonicalSet=NewType();
  TypeDenoter.GotType=
    ORDER(
      AddTypeToBlock(
        .CanonicalSet,
        SetTypes,
        SingleDefTableKeyList(TypeDenoter.Type)),
      ResetCanonicalSet(TypeDenoter.Type,.CanonicalSet));
  TypeDenoter.GotOper=
    ORDER(
      InstClass0(enumType,TypeDenoter.Type),
      InstClass1(setType,.CanonicalSet,TypeDenoter.Type),
      MonadicOperator(
        makeset,
```

```
        NoOprName,
        TypeDenoter.Type,
        FinalType(.CanonicalSet)));
  Enumerate.Type=TypeDenoter.Type;
END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

Property definitions[11]:

```
CanonicalSet: DefTableKey;
makeset;
```

This macro is defined in definitions 11, 19, 24, 30, 34, and 38.
This macro is invoked in definition 7.

3.1.3 Subrange types

Subrange types[12]:

```
CLASS rangeType(hostType) BEGIN
  OPER
    rangeOrd(rangeType): intType;
    rangeNarrow(hostType): rangeType;
  COERCION
    (rangeType): hostType;
END;

INDICATION
  ordType: rangeOrd;
  assignCvt: rangeNarrow;
```

This macro is invoked in definition 3.

Establish a user-defined type[13]:

```
RULE: TypeDenoter ::=  constant '..' constant COMPUTE
  .CanonicalSet=NewType();
  TypeDenoter.GotType=
    ORDER(
      AddTypeToBlock(
        .CanonicalSet,
        SetTypes,
        SingleDefTableKeyList(constant[1].Type)),
      ResetCanonicalSet(TypeDenoter.Type,.CanonicalSet));
  TypeDenoter.GotOper=
    InstClass1(rangeType,TypeDenoter.Type,constant[1].Type);
END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

3.1.4 Array types

Array types[14]:

```
CLASS arrayType(indexType,elementType) BEGIN
  OPER
    arrayaccess(arrayType,indexType): elementType;

/* FIXME: Array types can only be compared if they are packed 1..x of char
 **/
  COERCION (arrayType): stringType;
END;

OPER
  stringAccess(stringType): charType;
  stringcmp(stringType,stringType): boolType;

INDICATION
  arrayAccess: arrayaccess, stringAccess;
  equal: stringcmp;
  lsgt: stringcmp;
  less: stringcmp;
  greater: stringcmp;
  lessequal: stringcmp;
  greaterequal: stringcmp;
```

This macro is invoked in definition 3.

Establish a user-defined type[15]:

```
RULE: TypeDenoter ::=  'array' '[' type ']' 'of' type COMPUTE
  TypeDenoter.GotOper=
    InstClass2(arrayType,TypeDenoter.Type,type[1].Type,type[2].Type);
END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

3.1.5 Record types

Establish a user-defined type[16]:

```
ATTR OpndTypeList: DefTableKeyList;

SYMBOL Record INHERITS TypeDenotation, OperatorDefs END;

RULE: Record ::=  'record' Fields 'end' COMPUTE
  .Type=NewType();
  Record.GotType=
```

```
    AddTypeToBlock(
      .Type,
      PointerTypes,
      SingleDefTableKeyList(Record.Type));
  Record.GotOper=
    ORDER(
      InstClass1(ptrType,.Type,Record.Type),
      ListOperator(newType,NoOprName,Fields.OpndTypeList,voidType));
  Fields.OpndTypeList=SingleDefTableKeyList(.Type);
END;

SYMBOL var_part INHERITS OperatorDefs END;

RULE: var_part ::= 'case' var_sel 'of' Variants COMPUTE
  var_part.GotOper=
    ListOperator(newType,NoOprName,Variants.OpndTypeList,voidType);
  Variants.OpndTypeList=
    ConsDefTableKeyList(
      var_sel CONSTITUENT TypIdUse.Type,
      INCLUDING (Fields.OpndTypeList,Variants.OpndTypeList));
END;

TREE SYMBOL TagIdDef INHERITS TypedDefId END;

RULE: var_sel ::= TagIdDef ':' TypIdUse COMPUTE
  TagIdDef.Type=TypIdUse.Type;
END;

RULE: Variant ::= constants ':' '(' Fields ')' COMPUTE
  Fields.OpndTypeList=INCLUDING Variants.OpndTypeList;
END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

3.1.6 Set types

FIXME: This implementation makes all sets of a given base type equivalent. That's probably not correct. See Pascal 6.4.3.4, 6.7.2.4.

Set types[17]:

```
CLASS setType(baseType) BEGIN
  OPER
    setop(setType,setType): setType;
    setmember(baseType,setType): boolType;
    setrel(setType,setType): boolType;
```

```
  COERCION
    (emptyType): setType;
END;

INDICATION
  plus: setop;
  minus: setop;
  star: setop;
  in: setmember;
  equal: setrel;
  lsgt: setrel;
  lessequal: setrel;
  greaterequal: setrel;
```

This macro is invoked in definition 3.

Establish a user-defined type[18]:

```
RULE: TypeDenoter ::= 'set' 'of' type COMPUTE
  .CanonicalSet=
    GetCanonicalSet(FinalType(type.Type),NoKey)
    <- INCLUDING RootType.GotUserTypes;
  TypeDenoter.GotOper=
    ORDER(
      Coercible(
        NoOprName,
        FinalType(TypeDenoter.Type),
        FinalType(.CanonicalSet)),
      MonadicOperator(
        assignCvt,
        NoOprName,
        FinalType(.CanonicalSet),
        FinalType(TypeDenoter.Type)),
      InstClass1(setType,TypeDenoter.Type,type.Type));
END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

Property definitions[19]:

```
SetTypes;
```

This macro is defined in definitions 11, 19, 24, 30, 34, and 38.
This macro is invoked in definition 7.

3.1.7 File types

File types[20]:

```
CLASS fileType(componentType) BEGIN
  OPER
    filBuff(fileType): componentType;
    filOp(fileType): voidType;
    filTst(fileType): boolType;
END;

INDICATION
  deref:        filBuff;
  rewriteType:  filOp;
  putType:      filOp;
  resetType:    filOp;
  getType:      filOp;
  eofType:      filTst;
  eolnType:     filTst;
```

This macro is invoked in definition 3.

Establish a user-defined type[21]:

```
RULE: TypeDenoter ::=  'file' 'of' type COMPUTE
  TypeDenoter.GotOper=
    InstClass1(fileType,TypeDenoter.Type,type.Type);
END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

3.1.8 Pointer types

Pointer types[22]:

```
CLASS ptrType(domainType) BEGIN
  OPER
    ptrOrd(ptrType): intType;
    ptrderef(ptrType): domainType;
    ptreq, ptrne(ptrType,ptrType): boolType;
    ptrNewDisp(ptrType): voidType;
  COERCION
    (nilType): ptrType;
END;

INDICATION
  ordType: ptrOrd;
  deref: ptrderef;
  equal: ptreq;
  lsgt: ptrne;
  newType: ptrNewDisp;
  disposeType: ptrNewDisp;
```

This macro is invoked in definition 3.

Establish a user-defined type[23]:

```
RULE: TypeDenoter ::=  '^' type COMPUTE
  TypeDenoter.GotType=
    AddTypeToBlock(
      TypeDenoter.Type,
      PointerTypes,
      SingleDefTableKeyList(type.Type));
  TypeDenoter.GotOper=
    InstClass1(ptrType,TypeDenoter.Type,type.Type);
END;
```

This macro is defined in definitions 4, 10, 13, 15, 16, 18, 21, and 23.
This macro is invoked in definition 1.

Property definitions[24]:

```
PointerTypes;
```

This macro is defined in definitions 11, 19, 24, 30, 34, and 38.
This macro is invoked in definition 7.

3.2 Constant definitions

Constant definitions[25]:

```
CHAIN ConstDepend: VOID;

CLASS SYMBOL RootType COMPUTE
  CHAINSTART HEAD.ConstDepend = "yes";
END;

TREE SYMBOL ConIdDef  INHERITS TypedDefId      END;
TREE SYMBOL ConIdUse  INHERITS TypedUseId, ChkTypedUseId      END;
TREE SYMBOL Enumerate INHERITS TypedDefinition END;

RULE: Decl ::= ConIdDef '=' constant COMPUTE
  ConIdDef.Type=constant.Type;
  Decl.ConstDepend = ConIdDef.TypeIsSet <- constant.ConstDepend;
END;

SYMBOL ConIdUse COMPUTE
  SYNT.TypeIsSet=THIS.ConstDepend;
END;
```

This macro is defined in definitions 25 and 26.
This macro is invoked in definition 1.

Constant definitions[26]:

```
RULE: Literal ::= character_string COMPUTE
  Literal.Type=
    IF(EQ(strlen(StringTable(character_string)),3),charType,
    IF(strcmp(StringTable(character_string),"'''''"),stringType,
    charType));
END;

RULE: constant ::= csign integer_constant COMPUTE
  constant.Type=intType;
END;

RULE: constant ::= csign real_constant COMPUTE
  constant.Type=realType;
END;

RULE: constant ::= csign ConIdUse COMPUTE
  constant.Type=ConIdUse.Type;
END;

RULE: constant ::= integer_constant COMPUTE
  constant.Type=intType;
END;

RULE: constant ::= real_constant COMPUTE
  constant.Type=realType;
END;

RULE: constant ::= ConIdUse COMPUTE
  constant.Type=ConIdUse.Type;
END;

RULE: constant ::= Literal COMPUTE
  constant.Type=Literal.Type;
END;
```

This macro is defined in definitions 25 and 26.
This macro is invoked in definition 1.

3.3 Type definitions

Type definitions[27]:

```
TREE SYMBOL TypIdDef INHERITS TypeDefDefId, ChkTypeDefDefId END;
TREE SYMBOL TypIdUse INHERITS TypeDefUseId, ChkTypeDefUseId END;
```

```
RULE: type ::= TypIdUse COMPUTE
  type.Type=TypIdUse.Type;
END;

RULE: type ::= TypeDenoter COMPUTE
  type.Type=TypeDenoter.Type;
END;

RULE: type ::= Record COMPUTE
  type.Type=Record.Type;
END;
```

This macro is defined in definitions 27.
This macro is invoked in definition 1.

3.4 Variable declarations

A declaration is used to associated a type with an identifier, and that type becomes the value of the `Type` attribute of the identifier use. Thus the `Typing` module provides computational roles for both the defining occurrence and the applied occurrence of a typed identifier. These roles must be inherited by appropriate AST nodes:

Variable declarations[28]:

```
TREE SYMBOL VrblIds  INHERITS TypedDefinition END;
TREE SYMBOL FrmlIds  INHERITS TypedDefinition END;

TREE SYMBOL VblIdDef INHERITS TypedDefId      END;

RULE: Decl ::= TypIdDef '=' type     COMPUTE
  TypIdDef.Type=type.Type;
END;

RULE: Decl ::= VrblIds  ':' type     COMPUTE
  VrblIds.Type=type.Type;
END;
```

This macro is defined in definitions 28.
This macro is invoked in definition 1.

3.5 Procedure and function declarations

Procedure and function declarations[29]:

```
SYMBOL ProcHead INHERITS TypeDenotation, OperatorDefs END;
SYMBOL FuncHead INHERITS TypeDenotation, OperatorDefs END;
```

```
SYMBOL ProcBody INHERITS TypeDenotation, OperatorDefs END;
SYMBOL FuncBody INHERITS TypeDenotation, OperatorDefs END;

SYMBOL Formals  INHERITS OpndTypeListRoot END;
SYMBOL FmlIdDef INHERITS OpndTypeListElem END;

RULE: ProcHead ::= Formals COMPUTE
  ProcHead.GotType=
    AddTypeToBlock(
      ProcHead.Type,
      ProcTypes,
      Formals.OpndTypeList);
  ProcHead.GotOper=
    ListOperator(ProcHead.Type,NoOprName,Formals.OpndTypeList,voidType);
END;

RULE: ProcBody ::= Formals ';' block COMPUTE
  ProcBody.GotType=
    AddTypeToBlock(
      ProcBody.Type,
      ProcTypes,
      Formals.OpndTypeList);
  ProcBody.GotOper=
    ListOperator(ProcBody.Type,NoOprName,Formals.OpndTypeList,voidType);
END;

RULE: FuncHead ::= Formals ':' TypIdUse COMPUTE
  FuncHead.GotType=
    AddTypeToBlock(
      FuncHead.Type,
      FuncTypes,
      ConsDefTableKeyList(TypIdUse.Type,Formals.OpndTypeList));
  FuncHead.GotOper=
    ListOperator(FuncHead.Type,NoOprName,Formals.OpndTypeList,TypIdUse.Type);
END;

RULE: FuncBody ::= Formals ':' TypIdUse ';' block COMPUTE
  FuncBody.GotType=
    AddTypeToBlock(
      FuncBody.Type,
      FuncTypes,
      ConsDefTableKeyList(TypIdUse.Type,Formals.OpndTypeList));
  FuncBody.GotOper=
    ListOperator(FuncBody.Type,NoOprName,Formals.OpndTypeList,TypIdUse.Type);
END;
```

This macro is defined in definitions 29, 31, 32, and 33.
This macro is invoked in definition 1.

Property definitions[30]:

```
ProcTypes;
FuncTypes;
```

This macro is defined in definitions 11, 19, 24, 30, 34, and 38.
This macro is invoked in definition 7.

Functions with no arguments are invoked without an argument list. Since there is no syntactic clue, the "call" must be treated as a coercion.

Procedure and function declarations[31]:

```
RULE: FuncHead ::= ':' TypIdUse COMPUTE
  FuncHead.GotType=
    AddTypeToBlock(
      FuncHead.Type,
      FuncTypes,
      SingleDefTableKeyList(TypIdUse.Type));
  FuncHead.GotOper=Coercible(NoOprName,FuncHead.Type,TypIdUse.Type);
END;

RULE: FuncBody ::= ':' TypIdUse ';' block COMPUTE
  FuncBody.GotType=
    AddTypeToBlock(
      FuncBody.Type,
      FuncTypes,
      SingleDefTableKeyList(TypIdUse.Type));
  FuncBody.GotOper=Coercible(NoOprName,FuncBody.Type,TypIdUse.Type);
END;
```

This macro is defined in definitions 29, 31, 32, and 33.
This macro is invoked in definition 1.

Procedure and function declarations[32]:

```
CHAIN Forward: VOID;

CLASS SYMBOL RootType COMPUTE
  CHAINSTART HEAD.Forward = "yes";
END;

RULE: Decl ::= 'procedure' PrcIdDef ProcBody COMPUTE
  .Type=GetTypeOf(PrcIdDef.Key,NoKey) <- Decl.Forward;
  PrcIdDef.Type=IF(EQ(.Type,NoKey),ProcBody.Type,.Type);
```

```
  ProcBody.Forward=PrcIdDef.Type;
END;

RULE: Decl ::= 'function'  FncIdDef FuncBody COMPUTE
  .Type=GetTypeOf(FncIdDef.Key,NoKey) <- Decl.Forward;
  FncIdDef.Type=IF(EQ(.Type,NoKey),FuncBody.Type,.Type);
  FuncBody.Forward=FncIdDef.Type;
END;
```

This macro is defined in definitions 29, 31, 32, and 33.
This macro is invoked in definition 1.

Procedure and function declarations[33]:

```
TREE SYMBOL FmlIdDef INHERITS TypedDefId      END;
TREE SYMBOL PrcIdDef INHERITS TypedDefId      END;
TREE SYMBOL FncIdDef INHERITS TypedDefId      END;

RULE: Formal ::= FrmlIds ':' type COMPUTE
  FrmlIds.Type=type.Type;
END;

RULE: Formal ::= 'var' FrmlIds ':' type COMPUTE
  FrmlIds.Type=type.Type;
END;

RULE: Formal ::= 'procedure' FmlIdDef ProcHead COMPUTE
  FmlIdDef.Type=ProcHead.Type;
END;

RULE: Formal ::= 'function' FmlIdDef FuncHead COMPUTE
  FmlIdDef.Type=FuncHead.Type;
END;
```

This macro is defined in definitions 29, 31, 32, and 33.
This macro is invoked in definition 1.

Property definitions[34]:

```
absKey      -> TypeOf={absType};
arctanKey   -> TypeOf={arctanType};
chrKey      -> TypeOf={chrType};
cosKey      -> TypeOf={cosType};
disposeKey  -> TypeOf={disposeType};
eofKey      -> TypeOf={eofType};
eolnKey     -> TypeOf={eolnType};
expKey      -> TypeOf={expType};
getKey      -> TypeOf={getType};
```

```
inputKey    -> TypeOf={textType};
lnKey       -> TypeOf={lnType};
newKey      -> TypeOf={newType};
oddKey      -> TypeOf={oddType};
ordKey      -> TypeOf={ordType};
outputKey   -> TypeOf={textType};
predKey     -> TypeOf={predType};
putKey      -> TypeOf={putType};
readKey     -> TypeOf={readType};
readlnKey   -> TypeOf={readlnType};
resetKey    -> TypeOf={resetType};
rewriteKey  -> TypeOf={rewriteType};
roundKey    -> TypeOf={roundType};
sinKey      -> TypeOf={sinType};
sqrKey      -> TypeOf={sqrType};
sqrtKey     -> TypeOf={sqrtType};
succKey     -> TypeOf={succType};
truncKey    -> TypeOf={truncType};
writeKey    -> TypeOf={writeType};
writelnKey  -> TypeOf={writelnType};

absType     -> IsType={1};
arctanType  -> IsType={1};
chrType     -> IsType={1};
cosType     -> IsType={1};
disposeType -> IsType={1};
expType     -> IsType={1};
getType     -> IsType={1};
lnType      -> IsType={1};
newType     -> IsType={1};
oddType     -> IsType={1};
ordType     -> IsType={1};
predType    -> IsType={1};
putType     -> IsType={1};
readType    -> IsType={1};
readlnType  -> IsType={1};
resetType   -> IsType={1};
rewriteType -> IsType={1};
roundType   -> IsType={1};
sinType     -> IsType={1};
sqrtType    -> IsType={1};
sqrType     -> IsType={1};
succType    -> IsType={1};
truncType   -> IsType={1};
writeType   -> IsType={1};
writelnType -> IsType={1};
```

This macro is defined in definitions 11, 19, 24, 30, 34, and 38.
This macro is invoked in definition 7.

3.6 Operator.d

Operator.d[35]:

```
PreDefInd('=',   operator, equal)
PreDefInd('<>',  operator, lsgt)
PreDefInd('<',   operator, less)
PreDefInd('>',   operator, greater)
PreDefInd('<=',  operator, lessequal)
PreDefInd('>=',  operator, greaterequal)
PreDefInd('in',  operator, in)
PreDefInd('+',   operator, plus)
PreDefInd('-',   operator, minus)
PreDefInd('or',  operator, or)
PreDefInd('*',   operator, star)
PreDefInd('/',   operator, slash)
PreDefInd('div', operator, div)
PreDefInd('mod', operator, mod)
PreDefInd('and', operator, and)
PreDefInd('not', operator, not)
```

This macro is attached to a non-product file.

Expressions[36]:

```
TREE SYMBOL PrcIdUse INHERITS TypedUseId, ChkTypedUseId END;
TREE SYMBOL FncIdUse INHERITS TypedUseId, ChkTypedUseId END;
TREE SYMBOL ExpIdUse INHERITS TypedUseId, ChkTypedUseId END;
TREE SYMBOL FldIdUse INHERITS TypedUseId, ChkTypedUseId END;

SYMBOL expression    INHERITS ExpressionSymbol END;
SYMBOL variable      INHERITS ExpressionSymbol END;
SYMBOL Subscript     INHERITS ExpressionSymbol END;
SYMBOL operator      INHERITS OperatorSymbol   END;

RULE: expression ::= integer_constant COMPUTE
  PrimaryContext(expression,intType);
END;

RULE: expression ::= real_constant COMPUTE
  PrimaryContext(expression,realType);
END;
```

```
RULE: expression ::= Literal COMPUTE
  PrimaryContext(expression,Literal.Type);
END;

RULE: expression ::= 'nil' COMPUTE
  PrimaryContext(expression,nilType);
END;

RULE: expression ::= variable COMPUTE
  TransferContext(expression,variable);
END;

RULE: variable ::= ExpIdUse COMPUTE
  PrimaryContext(variable,ExpIdUse.Type);
END;

RULE: variable ::= variable '[' Subscript ']' COMPUTE
  DyadicContext(variable[1],,variable[2],Subscript);
  Indication(arrayAccess);
  IF(BadOperator,message(ERROR,"Invalid array reference",0,COORDREF));
END;

ATTR Sym: int;
ATTR env: Environment;
ATTR bnd: Binding;
ATTR ScopeKey, Key: DefTableKey;

RULE: variable ::= variable '.' FldIdUse COMPUTE
  variable[2].Required=NoKey <- FldIdUse.Type;
  PrimaryContext(variable[1],FldIdUse.Type);
END;

RULE: variable ::= variable '^' COMPUTE
  MonadicContext(variable[1],,variable[2]);
  Indication(deref);
  IF(BadOperator,message(ERROR,"Invalid pointer access",0,COORDREF));
END;

RULE: Subscript ::= expression COMPUTE
  ConversionContext(Subscript,,expression);
  Indication(assignCvt);
END;

SYMBOL Actual  INHERITS OpndExprListElem END;
SYMBOL FncArgs INHERITS OpndExprListRoot END;
```

```
RULE: expression ::= FncIdUse FncArgs COMPUTE
  ListContext(expression,,FncArgs);
  Indication(FncIdUse.Type);
  IF(BadOperator,message(ERROR,"Illegal function call",0,COORDREF));
END;

RULE: Actual ::= expression COMPUTE
  ConversionContext(Actual,,expression);
  Indication(assignCvt);
END;

SYMBOL PrcArgs INHERITS OpndExprListRoot END;

RULE: ProcCall ::= PrcIdUse PrcArgs COMPUTE
  ListContext(ProcCall,,PrcArgs);
  Indication(PrcIdUse.Type);
  IF(BadOperator,message(ERROR,"Illegal procedure call",0,COORDREF));
END;

/* FIXME: Need some type checking here */

RULE: InOutStmt ::= 'read' '(' RdArgs ')'
END;

RULE: InOutStmt ::= 'readln'
END;

RULE: InOutStmt ::= 'readln' '(' RdArgs ')'
END;

RULE: InOutStmt ::= 'write' '(' WrtArgs ')'
END;

RULE: InOutStmt ::= 'writeln'
END;

RULE: InOutStmt ::= 'writeln' '(' WrtArgs ')'
END;

RULE: WrtArg ::= expression ':' expression ':' expression COMPUTE
  expression[2].Required=intType;
  expression[3].Required=intType;
END;

RULE: WrtArg ::= expression ':' expression COMPUTE
  expression[2].Required=intType;
```

```
END;

RULE: WrtArg ::= expression COMPUTE
  Indication(assignCvt);
END;

SYMBOL Members INHERITS BalanceListRoot END;
SYMBOL Member  INHERITS BalanceListElem END;

RULE: expression ::= '[' Members ']' COMPUTE
  MonadicContext(expression,,Members);
  Indication(makeset);
  IF(BadOperator,message(ERROR,"No set with this base type",0,COORDREF));
END;

RULE: expression ::= '[' ']' COMPUTE
  PrimaryContext(expression,emptyType);
END;

RULE: Member ::= expression COMPUTE
  TransferContext(Member,expression);
END;

RULE: Member ::= expression '..' expression COMPUTE
  BalanceContext(Member,expression[1],expression[2]);
END;

RULE: expression ::= operator expression COMPUTE
  MonadicContext(expression[1],operator,expression[2]);
END;

RULE: expression ::= expression operator expression COMPUTE
  DyadicContext(expression[1],operator,expression[2],expression[3]);
END;
```

This macro is defined in definitions 36.
This macro is invoked in definition 1.

3.7 Statements

Statements[37]:

```
ATTR FuncName: DefTableKey;

RULE: statement ::= variable ':=' expression COMPUTE
  RootContext(
    IF(NE(variable.FuncName,NoKey),
```

```
      FinalType(GetResultType(variable.FuncName,NoKey)),
      variable.Type),
    ,
    expression);
  Indication(assignCvt);
END;

RULE: Decl ::= 'function' FncIdDef FuncBody COMPUTE
  FuncBody.FuncName=FncIdDef.Key;
END;

RULE: Decl ::= 'function' FncIdUse ';' Body COMPUTE
  Body.FuncName=FncIdUse.Key;
END;

RULE: FuncBody ::= Formals ':' TypIdUse ';' block COMPUTE
  FuncBody.GotResultType=ResetResultType(FuncBody.FuncName,TypIdUse.Type);
END;

RULE: FuncBody ::= ':' TypIdUse ';' block COMPUTE
  FuncBody.GotResultType=ResetResultType(FuncBody.FuncName,TypIdUse.Type);
END;

SYMBOL program COMPUTE
  SYNT.GotResultTypes=CONSTITUENTS FuncBody.GotResultType;
END;

SYMBOL variable COMPUTE
  SYNT.FuncName=NoKey <- INCLUDING RootType.GotAllTypes;
END;

SYMBOL program COMPUTE
  SYNT.FuncName=NoKey <- INCLUDING RootType.GotAllTypes;
END;

RULE: variable ::= ExpIdUse COMPUTE
  variable.FuncName=
    IF(EQ(
        ExpIdUse.Key,
        INCLUDING (FuncBody.FuncName,Body.FuncName,program.FuncName)),
      ExpIdUse.Key,
      NoKey);
END;
```

This macro is defined in definitions 37 and 39.

This macro is invoked in definition 1.

Property definitions[38]:

```
ResultType: DefTableKey;
```

This macro is defined in definitions 11, 19, 24, 30, 34, and 38.
This macro is invoked in definition 7.

Statements[39]:

```
RULE: statement ::= 'if' expression 'then' statement COMPUTE
  expression.Required=boolType;
END;

RULE: statement ::= 'if' expression 'then' statement 'else' statement COMPUTE
  expression.Required=boolType;
END;

RULE: statement ::= 'case' expression 'of' cases 'end' COMPUTE
END;

RULE: statement ::= 'repeat' StmtList 'until' expression COMPUTE
  expression.Required=boolType;
END;

RULE: statement ::= 'while' expression 'do' statement COMPUTE
END;

RULE: statement ::= 'for' ExpIdUse ':=' expression 'to' expression
                    'do' statement COMPUTE
END;

RULE: statement ::= 'for' ExpIdUse ':=' expression 'downto' expression
                    'do' statement COMPUTE
END;
```

This macro is defined in definitions 37 and 39.
This macro is invoked in definition 1.

3.8 Name analysis of qualified identifiers

The region that is the field specifier of a field identifier is excluded from the enclosing scopes. Thus an applied occurrence of a field identifier must identify a defining occurrence in a specific record. Moreover, the record's scope is obtained from the variable whose field is being accessed:

Qualified identifiers[40]:

```
TREE SYMBOL FldIdUse INHERITS IdentOcc, QualIdUse, ChkQualIdUse END;

RULE: variable ::= variable '.' FldIdUse COMPUTE
  FldIdUse.ScopeKey=variable[2].Type;
END;
```

This macro is defined in definitions 40, 42, and 43.
This macro is invoked in definition 1.

This requires another library module:

Instantiate appropriate modules[41]:

```
$/Name/ScopeProp.gnrc :inst
```

This macro is defined in definitions 41 and 44.
This macro is invoked in definition 2.

A record definition exports its field environment. The key carrying that environment is the type of the record:

Qualified identifiers[42]:

```
SYMBOL Record INHERITS ExportRange COMPUTE
  SYNT.ScopeKey=THIS.Type;
END;
```

This macro is defined in definitions 40, 42, and 43.
This macro is invoked in definition 1.

In a `WithBody`, a field identifier is indistinguishable from any other identifier. The `WithBody` is a region that inherits the record's environment:

Qualified identifiers[43]:

```
TREE SYMBOL WithBody INHERITS InhRange     END;
TREE SYMBOL WithVar  INHERITS InheritScope END;

RULE: statement ::= 'with' WithVar 'do' WithBody COMPUTE
  WithBody.GotInh=WithVar.InheritOk;
  WithVar.InnerScope=WithBody.Env;
END;

RULE: WithVar ::= variable COMPUTE
  WithVar.ScopeKey=variable.Type;
END;
```

This macro is defined in definitions 40, 42, and 43.
This macro is invoked in definition 1.

Instantiate appropriate modules[44]:

```
$/Name/AlgInh.gnrc :inst
```

This macro is defined in definitions 41 and 44.
This macro is invoked in definition 2.

4 Enforcing Constraints

ANSI/IEEE 770X3.97-1983 specifies a number of *constraints* upon the constructs of the language. If a program violates any of these constraints, it is illegal and an error report should be provided to the author. This section specifies tree computations that check for violation of the following constraints, and issue messages:

Context.lido[1]:

> *Pascal Section 6.2.2.1*[3]
> *Pascal Section 6.2.2.7*[4]

This macro is attached to a product file.

Most error reporting requires the Eli string concatenation module to construct error reports containing variable information; other modules will be instantiated later in this section as needed:

Context.specs[2]:

> `$/Tech/Strings.specs`
> *Instantiate appropriate modules*[5]

This macro is attached to a product file.

4.1 Pascal Section 6.2.2.1

Each identifier or label contained by the program-block shall have a defining-point. This context condition is checked by a module computation that is implemented by the `ChkIdUse` role:

Pascal Section 6.2.2.1[3]:

```
TREE SYMBOL LblIdUse INHERITS ChkIdUse END;
TREE SYMBOL ConIdUse INHERITS ChkIdUse END;
TREE SYMBOL TypIdUse INHERITS ChkIdUse END;
TREE SYMBOL PrcIdUse INHERITS ChkIdUse END;
TREE SYMBOL FncIdUse INHERITS ChkIdUse END;
TREE SYMBOL ExpIdUse INHERITS ChkIdUse END;
```

This macro is defined in definitions 3.
This macro is invoked in definition 1.

4.2 Pascal Section 6.2.2.7

When an identifier or label has a defining point for a region, another identifier or label with the same spelling can't have a defining point for that region.

Pascal Section 6.2.2.7[4]:

```
SYMBOL MultDefChk INHERITS Unique COMPUTE
  IF(NOT(THIS.Unique),
    message(
      ERROR,
      CatStrInd("Multiply defined identifier ",THIS.Sym),
      0,
      COORDREF));
END;

TREE SYMBOL LblIdDef INHERITS MultDefChk END;
TREE SYMBOL ConIdDef INHERITS MultDefChk END;
TREE SYMBOL TypIdDef INHERITS MultDefChk END;
TREE SYMBOL TagIdDef INHERITS MultDefChk END;
TREE SYMBOL VblIdDef INHERITS MultDefChk END;
TREE SYMBOL FmlIdDef INHERITS MultDefChk END;
```

This macro is defined in definitions 4 and 6.
This macro is invoked in definition 1.

`MultDefChk` requires the Eli `Unique` module.

Instantiate appropriate modules[5]:

```
$/Prop/Unique.gnrc :inst
```

This macro is defined in definitions 5.
This macro is invoked in definition 2.

Function and procedure identifiers present a problem because of the `forward` directive. FIXME For the moment, no messages will be issued for multiply-defined procedures or functions.

Pascal Section 6.2.2.7[6]:

```
TREE SYMBOL PrcIdDef INHERITS IdentOcc, IdDefScope END;
TREE SYMBOL FncIdDef INHERITS IdentOcc, IdDefScope END;
```

This macro is defined in definitions 4 and 6.
This macro is invoked in definition 1.

Program parameters have similar characteristics, except that they must have defining occurrences a variables in the `program`. The required identifiers `input` and `output` are exceptions to this rule — their appearance in `PgmPars` constitutes their defining occurrence. FIXME That subtlety is not reflected in this specification.

FIXME This rather simplistic solution doesn't worry about multiple `forward` or non-`forward` declarations for the same routine. Strictly speaking, `input` and `output` should not be included in this list, since they are variables. A variable requires storage, and hence must have a defining point within the program. Defining them here means that there will be no error report if these identifiers are defined explicitly in the program block.