ALGOL 60

W. M. Waite

December 1, 2012

Abstract

This document is an Eli specification from which an analyzer for ALGOL 60 can be generated, or which can be used as one component of a complete ALGOL 60 compiler. Its structure mirrors that of the ALGOL 60 Revised Report, providing traceability and easing maintenance.

The specification was originally developed in 1995 as a class project at the University of Colorado by J. Amundsen, B. D. Basham, S-C. Chiang, D. A. Ence, R. K. Hill, J. E. Kvamme, O. Lokkebo, H. Ma, R. S. Matthieu, M. T. Rupawalla and W. Wang. It was put into its present form while the author was a Visiting Fellow at Australian National University in 1996. A major rewrite in 2012 simplified the tree structure and updated the tree computations to make use of the Eli type analysis modules.

Contents

1 Introduction

This document was generated by the Eli system from a specification module that provides a complete description of ALGOL 60. ALGOL 60 was defined in the classic paper "Revised Report on the Algorithmic Language ALGOL 60", which appeared (among other places) in the January, 1963 issue of *Communications of the ACM* (pages 1-17). We have structured the specification module to correspond as closely as possible to that paper, so that it is easy for the reader to verify that the specification module describes the same language as the paper.

From this specification module, the Eli system can generate an executable program that checks its input to verify conformance to the rules of ALGOL 60. This specification module can also be combined with other specification modules to define additional analysis of the input program and/or translation of the algorithm embodied in that program to another language.

This specification defines a hardware language that differs from the reference language of the Revised Report in some of the symbols used to represent objects. These differences will be pointed out in the appropriate sections of this document.

We have also restricted the language in two ways: unsigned integers cannot be used as labels, and type specifications must be given for all procedure parameters. The former restriction avoids a known ambiguity, and the latter is necessary for compile-time determination of the types of expressions. Both are common in ALGOL 60 implementations (Randell, B., Russell, L. J. "ALGOL 60 Implementation").

1.1 Formalism for Syntactic Description

We have used a more modern form of BNF notation in which the quoting conventions are reversed: The Revised Report uses angle brackets to quote nonterminal symbols, and leaves terminal symbols unquoted; we leave nonterminal symbols unquoted and use apostrophes to quote terminal symbols.

One consequence of this quoting strategy is that spaces are not allowed within nonterminal names. In order to preserve the understandability of the nonterminal names used in the Revised Report, we have adopted the convention of running the words together and capitalizing the first letter of every word. Thus we render the Revised Report's nonterminal `<simple variable>` as `SimpleVariable`.

The Revised Report uses a combination of BNF and English to describe the complete syntax of the language. Because this is an executable specification, we follow standard compiler practice, using BNF to describe the phrase structure and using a combination of regular expressions and C code to describe the basic symbols and comments.

1.1.1 Phrase Structure

A type-`con` file provides the definition of the context-free grammar to be used by the parser to recognize the structure of a program.

algol60.con[1] ≡
 Logical Values[26]
 Numbers[46]
 Variables[54]
 Function Designators[59]
 Expressions[53]
 Compound Statements and Blocks[90]
 Assignment Statements[93]
 Go To Statements[100]
 Dummy Statements[103]
 Conditional Statements[105]
 For Statements[107]
 Procedure Statements[111]

Declarations[125]

This macro is attached to a product file.

1.1.2 Basic Symbols and Comments

The definition of a basic symbol or comment consists of a *regular expression*, an *auxiliary scanner*, and a *token processor*. Either the auxiliary scanner or the token processor, or both, may be omitted.

The regular expression defines a sequence of characters that identifies the particular basic symbol or comment in the text. If this character sequence does not constitute the complete basic symbol or comment, then an auxiliary scanner that obtains the complete sequence is specified. Finally, if additional actions are required after the sequence has been recognized, then a token processor is specified to carry out those actions. Auxiliary scanners and token processors for common situations can be found in the Eli library. Routines to handle special situations can be written in C or C++ and provided as part of the specification.

For example, consider an ALGOL 60 string. This basic symbol is identified by the opening string quote ‘. A single opening string quote does not constitute the complete string, and the nested structure defined in Section 2.6.1 of the Revised Report cannot be described by a regular expression. Therefore we have written a C-coded auxiliary scanner named `Algol60String` to obtain the complete sequence. After the string has been recognized, an internal representation must be created. The token processor `mkidn` from the Eli library can be used to carry out that task.

A type-`gla` file provides the definitions of the basic symbols and comments. Each definition consists of a regular expression and (if necessary) the names of an auxiliary scanner and/or a token processor.

algol60.gla[2] ≡
Delimiters[28]
Unsigned Numbers[45]
Identifiers[35]
Strings[47]
Parameter Delimiter Letter String[60]
Code as Procedure Body[156]

This macro is attached to a product file.

A type-`c` file contains the code of the auxiliary scanners and token processors that are written to support the specification.

algol60.c[3] ≡

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "err.h"
#include "gla.h"
#include "source.h"
#include "tabsize.h"
#include "litcode.h"

```

Comment Scanner[31]
Token processors for comment delimiters[29]

String Scanner[48]

This macro is attached to a product file.

All auxiliary scanners obey the following interface specification:

Define an Auxiliary Scanner[4](◇1) ≡

```
char *
◇1(char *start, int length)
/* Standard interface for an auxiliary scanner
 *   On entry-
 *     start points to the first character of the scanned string
 *     length=length of the scanned string
 *   On exit-
 *     The function returns a pointer to the first character
 *       beyond the scanned string
 ***/
```

This macro is invoked in definitions 31 and 48.

All token processors obey the following interface specification:

Define a token processor[5](◇1) ≡

```
void
◇1(const char *start, int length, int *syncode, int *intrinsic)
/* Token processor "Name"
 *   On entry-
 *     start points to the first character of the sequence being
 *     classified
 *     length=length of the sequence being classified
 *     syncode points to a location containing the initial
 *     classification
 *     intrinsic points to a location to receive the value
 *   On exit-
 *     syncode points to a location containing the final
 *     classification
 *     intrinsic points to a location containing the value (if
 *     relevant)
 ***/
```

This macro is invoked in definitions 29 and 34.

1.2 Formalism for Semantically-Equivalent Symbols

Many distinct symbols used in the syntax of the Revised Report refer to semantically-identical constructs. We group such symbols into equivalence classes, and use a single symbol in the semantic description. This reduces the size of the semantic description, and emphasizes the uniformity of meaning among the equivalent symbols.

algol60.map[6] ≡

```
MAPSYM
```
Equivalences[55]

This macro is attached to a product file.

Our specifications of equivalence classes have replaced the examples of ALGOL 60 code that appear in the Revised Report. Where no equivalence classes are defined, we have inserted an empty section in order to keep the section numbering parallel to that of the Revised Report.

1.3 Formalism for Semantic Description

We use an attribute grammar to describe the semantics of ALGOL 60 via a series of tree computations. The description specifies only the relationships that must hold among these computations; Eli can deduce the order in which the computations should be carried out, and decide on how (or indeed whether) to store intermediate values.

Relationships among computations are embodied in *attributes*, which constitute pre- and post-conditions for computations. An attribute may have a value, and that value may be of any type. Types are often defined by external modules that export a set of appropriate operations for use in computation. The pre- and post-conditions may relate computations locally, or remotely along root-to-leaf or depth-first, left-to-right paths.

Eli's attribute grammar notation is LIDO. A LIDO specification consists of a set of symbol and rule computations, with additional definitions of the types of attributes. All computations must be expressed as function application. (Non-strict functions are available for dealing with conditional computation.) Computations may be associated with specific contexts in the tree (*rule* computations), or with every context in which a particular symbol appears (*symbol* computations).

algol60.lido[7] ≡
Internal representation of an identifier[36]
Binding Identifier Uses to their Definitions[8]
Scope Violation Rules[37]

Semantics[51]

Subscripts[57]

Type Attributes[52]
Establish operator indications for source symbols[73]
Identify an operator from its context[14]
Balance the operands of a conditional[15]

Verify type consistency of a left-part list[97]
Verify that an expression can be assigned to a left-part list[98]
The for List Elements[110]
Actual-Formal Correspondence[116]
Restrictions[120]
Values of Function Designators[149]
Specifications[152]
Lower upper bound expressions[137]

This macro is attached to a product file.

1.3.1 Binding Identifier Uses to their Definitions

An identifier may have different meanings in different parts of an ALGOL 60 program. We deal with this by using a *definition table key* to represent the meaning of an identifier internally. Definition table keys are unique handles providing access to the specific set of properties that characterize an entity.

Binding Identifier Uses to their Definitions[8] ≡

```
ATTR Key: DefTableKey;
```

This macro is invoked in definition 7.

Identifiers are freely chosen by the programmer and given meaning through specific language constructs. The general strategy is to bind the definition of an identifier and all of its uses to the same definition table key. When processing a construct that gives meaning to an identifier, properties characterizing that meaning are associated with the definition table key to which the definition is bound, and when processing a construct that uses an identifier those properties are accessed.

Eli provides modules for establishing these bindings. Each module implements a specific set of rules governing the binding process, and any number of copies of any of the available modules can be instantiated in a given specification.

Instantiate the ALGOL 60 consistent renaming module[9] ≡
```
$/Name/AlgScope.gnrc :inst
```

This macro is invoked in definition 19.

1.3.2 Verifying Type Consistency

ALGOL 60 places a number of context conditions on expressions, and those conditions are defined formally in this document. The tree computations to determine the type of each expression and to verify the context conditions are embodied in two Eli modules:

Instantiate the type analysis modules[10] ≡
```
$/Type/Typing.gnrc :inst
$/Type/Expression.gnrc :inst
```

This macro is invoked in definition 19.

For each operator, three steps are required: an *operator indication* must be established for the operator's source language symbol, one or more *operators* must be associated with the indication, and the *signatures* of those operators must be defined.

Operator indications are established for each operator's source language symbol by computations described in LIDO:

Set a monadic operator indication[11](⋄2) ≡
```
RULE: Uniop ::= '⋄1' COMPUTE Uniop.Indic=⋄2; END;
```

This macro is invoked in definitions 73 and 84.

Set a dyadic operator indication[12](⋄2) ≡
```
RULE: Binop ::= '⋄1' COMPUTE Binop.Indic=⋄2; END;
```

This macro is invoked in definitions 73, 75, 77, 84, and 86.

Signatures for the operators are described, and operators associated with indications, by descriptions written in OIL:

algol60.oil[13] ≡
Define operators and associate them with indications[67]
Define an operator to be applied when needed[131]
Define operators to establish types without values[102]

This macro is attached to a product file.

OIL *describes* a set of constraints that the operators and expressions must satisfy. Those constraints must be *verified* by tree computations that use computational roles exported by the type analysis modules.

Identify an operator from its context[14] ≡
```
SYMBOL Expression INHERITS ExpressionSymbol END;
SYMBOL Uniop      INHERITS OperatorSymbol   END;
SYMBOL Binop      INHERITS OperatorSymbol   END;

RULE: Expression ::= Uniop Expression COMPUTE
  MonadicContext(Expression[1],Uniop,Expression[2]);
END;

RULE: Expression ::= Expression Binop Expression COMPUTE
  DyadicContext(Expression[1],Binop,Expression[2],Expression[3]);
END;
```
This macro is invoked in definition 7.

The constraint on a conditional expression is that the type of the result must not depend on the (run-time) outcome of the condition:

Balance the operands of a conditional[15] ≡
```
RULE: Expression ::= IfClause Expression 'else' Expression COMPUTE
  BalanceContext(Expression[1],Expression[2],Expression[3]);
END;
```
This macro is invoked in definition 7.

1.4 Formalism for Properties of Quantities

Each quantity is represented internally by a definition table key, which is used to access the properties of that quantity. Properties are declared individually in a type-`pdl` file:

algol60.pdl[16] ≡
Property characterizing all quantities[50]
Dimension property of an array[135]
Properties characterizing function designators[150]
Properties characterizing formal parameters[115]
Formal parameter key list[147]
Property to support Section 5.2.4.2 check[139]
Property to support Section 4.7.5.2 check[121]

This macro is attached to a product file.

The declaration of a property specifies the type of value that property can hold. All C basic types, plus the type `DefTableKey` (the type of a definition table key) can be used without further specification; any other types must be specified by some type-`h` file whose name is given in the type-`pdl` file.

1.5 Formalism for Modules and Interfaces

Computations that verify context conditions are often expressed in terms of abstract data types that are exported by modules either defined by the user or obtained from Eli's library.

1.5.1 User-Defined Modules

A type-`h` file is used to define the interfaces for user-defined modules:

algol60.h[17] ≡
```
#ifndef ALGOL60_H
#define ALGOL60_H

```
Kinds of quantities[49]
Parameter identification[118]
Monitoring interface[20]

```
#endif
```

This macro is attached to a product file.

Because type-`h` files may be included by several different specifications, they must be protected against multiple inclusion. Eli's convention is to use a symbol consisting of the file name rendered in upper case, with periods replaced by underscores.

The definitions must be made available to the routines that implement the computations, by including them in a type-`head` file:

algol60.head[18] ≡
```
#include "algol60.h"
```

This macro is attached to a product file.

1.5.2 Library Modules

Library modules are instantiated by Eli requests named in a type-`specs` file:

algol60.specs[19] ≡
Instantiate the ALGOL 60 consistent renaming module[9]
Instantiate modules to pre-define identifiers[65]
Instantiate the type analysis modules[10]
Instantiate modules to support MultDefChk[38]
Instantiate modules for list handling[117]
Instantiate additional modules for uniqueness checks[153]

This macro is attached to a product file.

1.5.3 Monitoring interface

This specification defines two types, `KindOfQuantity` (Section 2.7) and `Parameter` (Section 4.7.4). It is useful when debugging the specification to be able to examine the values of these types, so we need to specify a monitoring interface for them.

Monitoring interface[20] ≡
```
#ifdef MONITOR
#define DAPTO_RESULTKindOfQuantity(k) DAPTO_RESULT_INT (k)
```

```
#define DAPTO_RESULTParameter(k) DAPTO_RESULT_PTR (k)
#define DAPTO_ARGParameter(k) DAPTO_ARG_PTR (k, Parameter)
#endif
```

This macro is invoked in definition 17.

In order to print the monitored values, we need some TCL code.

algol60.tcl[21] ≡
Print KindOfQuantity values[22]
Print Parameter values[23]

This macro is attached to a product file.

`KindOfQuantity` is an enumerated type, and it is useful to print the identifiers of that type rather than simply the index.

Print KindOfQuantity values[22] ≡
```
set n(KindOfQuantity,desc) "Kind of quantity represented"

proc n_KindOfQuantity_say {i} {
    set l {UndefinedIdn VariableIdn ArrayIdn LabelIdn SwitchIdn ProcedureIdn}
    n_say "KindOfQuantity:[lindex $l $i]"
}
```

This macro is invoked in definition 21.

A `Parameter` is a definition table key, so we can use the monitor print routines already available for type `DefTableKey`.

Print Parameter values[23] ≡
```
set n(Parameter,desc) "Internal parameter name"

proc n_Parameter_open {text key} {
    n_DefTableKey_open $text $key
}

proc n_Parameter_say {key} {
    n_DefTableKey_say $key
}
```

This macro is invoked in definition 21.

2 Basic Symbols, Identifiers, Numbers, and Strings

The basic symbols of our specification language are identifiers, denotations and delimiters rather than the characters making them up.

Basic symbols are recognized by a finite-state machine, and therefore the definitions of the basic symbols are stated as regular expressions rather than productions of a context-free grammar. This means that all recursive definitions given in the Revised Report must be replaced by the equivalent iterative formulations. We have kept the regular expression notation as close to the form of the context-free grammar as possible to ease verification, except that we have replaced alternations with a more compact representation when

describing sets of characters: `[a-d]` means `a|b|c|d` and `[^;]` means "any character other than semicolon", for example.

The scanner that recognizes basic symbols attaches source text coordinates (the line number and column number of the first character) to each basic symbol. This requires that the scanner keep track of the coordinates as it is scanning. Coordinate tracking is handled automatically for most basic symbols, but must be provided explicitly for basic symbols that may contain horizontal tab or newline characters. These cases are noted below when they occur.

2.1 Letters

letter[24] ≡
```
[a-zA-Z]
```
This macro is invoked in definition 35.

Letters are used only as components of other basic symbols.

2.2

2.2.1 Digits

digit[25] ≡
```
[0-9]
```
This macro is invoked in definitions 35 and 39.

Digits are used only as components of other basic symbols.

2.2.2 Logical Values

Logical Values[26] ≡
```
LogicalValue ::= 'true' / 'false' .
```
This macro is invoked in definition 1.

The logical values are denoted by keywords of the language.

2.3 Delimiters

Most delimiters appear as quoted literals in the grammar, so their definitions need not be repeated here. The exceptions are those delimiters associated with comments: `;`, `begin`, and `end`.

We still want to represent these delimiters as quoted literals in the grammar, but we need to specify that they must be processed specially:

algol60.delit[27] ≡
```
$;      Semi
$begin  begin
$end    end
```
This macro is attached to a product file.

This specification requires that we recognize each of the delimiters using appropriate scanners, and then use a token processor to set the `syncode` to `Semi`, `begin`, or `end`. The actual text of each delimiter is specified as usual, except that no symbol is associated with the scan because the symbol will be supplied by the token processor.

Delimiters[28] ≡
```
$;([\040\t\n]*comment[^;]*;)?      (coordAdjust)        [SemiCmt]
$begin([\040\t\n]+comment[^;]*;)?  (coordAdjust)        [BeginCmt]
```

This macro is defined in definitions 28 and 30.
This macro is invoked in definition 2.

(`\040` must be used to represent a space within a regular expression because *any* white space terminates the expression.)

Token processors for comment delimiters[29] ≡
```
    Define a token processor[5](‘SemiCmt’)
    { *syncode = Semi; }

    Define a token processor[5](‘BeginCmt’)
    { *syncode = begin; }
```

This macro is defined in definitions 29 and 34.
This macro is invoked in definition 3.

Both `Semi` and `begin` are easy to define with regular expressions because they are self-delimiting. The regular expression accepts horizontal tabs and newlines, and therefore the auxiliary scanner `coordAdjust` must be used to adjust the coordinates appropriately.

Comments following `end` are much harder to deal with because they can *not* contain certain strings. It would be possible to write a regular expression that matched strings ending in `end` or `;` or `else`, and then use an auxiliary scanner to accept everything up to but not including that symbol. The problem is that the scanner would then find the *longest* such string, and we need the *shortest* such string.

Thus we need to use an auxiliary scanner to pick up the comment string following an `end`:

Delimiters[30] ≡
```
$end    (Algol60Comment)        [EndCmt]
```

This macro is defined in definitions 28 and 30.
This macro is invoked in definition 2.

An auxiliary scanner is invoked after the associated regular expression has been accepted. In this case, `Algol60Comment` will be invoked after the normal scanner has accepted the symbol `end`.

The auxiliary scanner is called with a pointer to the first character matched by the regular expression and the length of the string matching the regular expression; it should return a pointer to the first character that does *not* belong to the basic symbol being recognized. Thus `start` should point to the `e` of `end`, `length` should be 3, and `Algol60Comment` should return a pointer to the next `end`, `;` or `else`.

The regular expression will match the characters `e`, `n` and `d` in that order. Since the scanner finds the longest match, the character following the `d` cannot be a letter or a digit. If it were, the scanner would combine it with the first three characters in searching for an `identifier`. The variable `p` is set to point to the character following the `d`, so after fetching that character to `c` and incrementing `p` the stated loop invariant holds:

Comment Scanner[31] ≡
```
    Define an Auxiliary Scanner[4](‘Algol60Comment’)
```

```
{ char *p = start + length;
  char c = *p++;

  for (;;) {    /* Invariant:
                 *   c is neither a letter nor a digit &&
                 *   p points to the first unexamined character
                 * Bound: Number of characters left in the file
                 ***/

    if (c == ';') return p - 1;

    Adjust coordinates if necessary[32](‘’)

    if (*p == 'e') {
        if (strncmp(p, "end", 3) == 0 && !isalnum(p[3]) ||
            strncmp(p, "else", 4) == 0 && !isalnum(p[4]))
          return p;
    }

    while (isalnum(c = *p++)) ;
  }
}
```

This macro is defined in definitions 31.
This macro is invoked in definition 3.

If `c` is `;`, then the scan should be terminated and the position of the semicolon returned. Otherwise, any necessary adjustment of the coordinates to reflect a tab or newline character must be made.

The first unexamined character, pointed to by `p`, could be anything. If it is an `e`, then the routine checks for the presence of one of the two keywords that can follow the comment. Each of the specified character sequences must be followed by a character other than a letter or digit if it is to be recognized as a keyword. When either of those keywords is found, the routine returns the position of that keyword. (Thus terminating the comment with the character preceding the keyword.)

Finally, the routine re-establishes the invariant of the outer loop.

The generated scanner keeps track of the line and column number of each basic symbol, so that the processor can report errors at the appropriate position in the source program. Tab and newline characters require special action to maintain these values:

Adjust coordinates if necessary[32](◇1) ≡

```
if (c == '\t') StartLine -= TABSIZE(p - StartLine);
else if (c == '\n') {
  Refill the source buffer if necessary[33](‘◇1’)
  LineNum++;
  StartLine = p - 1;
}
```

This macro is invoked in definitions 31 and 48.

Character position within a line is the difference between the current character index and the character index `StartLine`. A tab character occupies one character position, but it may represent a longer sequence, depending upon it's position in the line. This additional white space is taken into account by moving `StartLine` backwards.

The coordinate adjustment code is needed for processing strings (Section 2.6) as well. These two cases differ in that it is an error for the file to end within a string, but no error if the file ends within a comment. If an end-of-file is an error, code to report that error must be introduced. That code is introduced through the parameter to the coordinate adjustment description.

Source text is stored as a sequence of characters in memory, and the last character of that sequence is guaranteed to be a newline. If the character following a newline is the ASCII NUL character, then there are no more characters in the sequence; otherwise, this newline is not the last newline of the sequence. Thus when the scanner reaches a newline, it must check whether that newline is the last of the sequence in memory:

Refill the source buffer if necessary[33](◇1) ≡

```
if (*p == '\0') {
  size_t pSave = p - start, sSave = p - StartLine;

  refillBuf(start); TokenStart = start = TEXTSTART;
  p = start + pSave; StartLine = start + sSave;

  if (*p == '\0') {
    ◇1
    return p - 1;
  }
}
```

This macro is invoked in definition 32.

Recall that `p` points to the character following the newline. If it is the ASCII NUL character, then there are no more characters of the file stored in memory. By calling `refillBuf` with `start` as an argument, the scanner *extends* the character sequence that begins at the location pointed to by `start`. Because of the way storage is allocated by the source text input module, it may be necessary to move the sequence to another location. Thus `TokenStart` (the starting location of the basic symbol), `start`, and `p` must all be adjusted. Finally, a check is made to see whether the character sequence was, in fact, extended. If it was not, then end-of-file has been reached, the comment string has ended, and the scanner terminates.

Token processors for comment delimiters[34] ≡

```
Define a token processor[5]('EndCmt')
{ *syncode = end; }
```

This macro is defined in definitions 29 and 34.
This macro is invoked in definition 3.

2.4 Identifiers

Identifiers[35] ≡

```
Identifier:      $letter[24](letter[24]|digit[25])*        [mkidn]
```

This macro is invoked in definition 2.

This regular expression is equivalent to the definition given in the Revised Report.

Distinct identifiers must be distinguishable from one another. The token processor `mkidn` assigns a unique integer value to each distinct identifier. This integer value can also be used to access the string representing the identifier, and is stored as the value of the `Sym` attribute of symbols representing identifier occurrences:

Internal representation of an identifier[36] ≡
```
ATTR Sym: int;

CLASS SYMBOL IdentOcc COMPUTE SYNT.Sym = TERM; END;
```
This macro is invoked in definition 7.

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures.

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program:

Scope Violation Rules[37] ≡
```
SYMBOL MultDefChk INHERITS Unique COMPUTE
  IF(NOT(THIS.Unique),
    message(
      ERROR,
      CatStrInd("Identifier is multiply defined: ", THIS.Sym),
      0,
      COORDREF));
END;
```
This macro is defined in definitions 37, 126, and 127.
This macro is invoked in definition 7.

`Unique` and `CatStrInd` are exported by Eli library modules.

Instantiate modules to support MultDefChk[38] ≡
```
$/Prop/Unique.gnrc :inst
$/Tech/Strings.specs
```
This macro is invoked in definition 19.

2.5 Numbers

2.5.1 Syntax

unsigned integer[39] ≡
digit[25]`+`This macro is invoked in definitions 40, 41, 43, and 45.

This regular expression is equivalent to the definition given in the Revised Report.

integer[40] ≡
`(`*unsigned integer*[39]`|\+`*unsigned integer*[39]`|-`*unsigned integer*[39]`)`This macro is invoked in definition 42.

The `+` must be escaped because it is a regular expression operator.

decimal fraction[41] ≡
`\.`*unsigned integer*[39]This macro is invoked in definition 43.

The `.` must be escaped because it is a regular expression operator.

Following standard practice, we represent the subscript 10 of the reference language by the letter `E`:

exponent part[42] ≡
`(E|e)` *integer*[40] This macro is invoked in definitions 44 and 45.

By requiring a `decimal number` to contain a `decimal fraction`, we obtain distinct representations for `integer` and `real` constants:

decimal number[43] ≡
`(` *decimal fraction*[41] `|` *unsigned integer*[39] *decimal fraction*[41] `)` This macro is invoked in definition 44.

We cannot allow an `unsigned number` consisting of only an `exponent part`, because there would be no way for the scanner to distinguish it from an identifier (this is a consequence of our representing the subscript 10 of the reference language by the letter `E`):

unsigned number[44] ≡
`(` *decimal number*[43] `|` *decimal number*[43] *exponent part*[42] `)` This macro is invoked in definition 45.

We express the distinction between `integer` and `real` constants by introducing the new basic symbol `UnsignedReal`, which does not appear in the Revised Report:

Unsigned Numbers[45] ≡
```
UnsignedInteger: $unsigned integer[39]                                    [mkidn]
UnsignedReal:    $unsigned number[44] | unsigned integer[39] exponent part[42] [mkidn]
```
This macro is invoked in definition 2.

The token processor `mkidn` assigns a unique integer value to each distinct numeric denotation. (Note that distinct denotations may represent the same numeric value: `001` has the same value as `1` and `1.23E1` has the same value as `12.3`.) This integer value can be used to access the denoting string.

An `unsigned integer` followed by an `exponent part` is legal according to the Revised Report. Because of the need to exclude `unsigned integer` from the definition of `decimal number`, however, this case has to be treated specially.

Numbers[46] ≡
```
UnsignedNumber ::= UnsignedInteger / UnsignedReal .
```
This macro is invoked in definition 1.

The symbol `number` does not appear elsewhere in the Revised Report, so it is not defined here.

2.6 Strings

The Revised Report's definition of a string allows arbitrary nesting of character sequences delimited by ‘ and ’. Such a structure cannot be described by a finite-state machine, and therefore an auxiliary scanner must be used:

Strings[47] ≡
```
String:          $‘    (Algol60String)    [mkidn]
```
This macro is invoked in definition 2.

`Algol60String` will be invoked after the normal scanner has accepted the character ‘.

String Scanner[48] ≡

Define an Auxiliary Scanner[4](‘`Algol60String`’)

```
{ char *p = start + length;
  int counter = 1;

                /* Invariant:
                 *   counter = number of unmatched open quotes
                 *   p points to the first unexamined character
                 * Bound: Number of characters left in the file
                 ***/
  while (counter) {
    char c = *p++;

    Adjust coordinates if necessary[32](‘message(ERROR,"End-of-file in string",0,&curpos);’)

    if (c == ’\’’) counter--; else if (c == ’‘’) counter++;
  }

  return p;
}
```

This macro is invoked in definition 3.

2.7 Quantities, Kinds and Scopes

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

Kinds of quantities[49] ≡

```
typedef enum {
  UndefinedIdn,
  VariableIdn,
  ArrayIdn,
  LabelIdn,
  SwitchIdn,
  ProcedureIdn
} KindOfQuantity;
```

This macro is invoked in definition 17.

`UndefinedIdn` is used to distinguish identifiers that are used but not declared. Every declared quantity is represented internally by a definition table key under which the kind is stored as a property:

Property characterizing all quantities[50] ≡

```
Kind: KindOfQuantity;   "algol60.h"
```

This macro is defined in definitions 50.
This macro is invoked in definition 16.

The kind of quantity may also be stored as an attribute of certain nodes, and a dependence must be used to guarantee that all quantities have their `Kind` properties set before any are queried:

Semantics[51] ≡

```
ATTR kind: KindOfQuantity;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

2.8 Values and Types

Certain syntactic units are said to possess values. The various "types" (`integer`, `real`, `Boolean`) basically denote properties of these values.

The Revised Report does not consider that labels and strings have values, but this position is inconsistent with the fact that labels and strings can be passed as parameters. For uniformity, therefore, we specify operators for labels and strings as well as for `integer`, `real` and `Boolean` values. These specifications result in the values `labelType` and `stringType`, which we use to represent the "types" of labels and strings.

Some procedures do not yield values, but may have side effects on the state of the computation. Our specifications of the operators characterizing these procedures return a "value" of type `voidType`.

Type Attributes[52] ≡
```
ATTR Type: DefTableKey;
```
This macro is invoked in definition 7.

3 Expressions

The grammar given in the Revised Report is ambiguous. Several changes were needed to eliminate the ambiguity; these will be pointed out in the appropriate sections.

Because type information is required to distinguish arithmetic, Boolean and designational expressions, we have merged the grammars for these three kinds of expression. Wherever possible, however, we have retained the nonterminal symbols used in the Revised Report.

The Revised Report distinguishes various classes of identifier: variable identifier, array identifier, switch identifier and procedure identifier. Without information about how an identifier was declared, it is impossible to make this distinction in all contexts. We have therefore replaced some occurrences of the nonterminals variable identifier, array identifier, switch identifier and procedure identifier with the terminal `Identifier`.

Expressions[53] ≡
Arithmetic Expressions[70]
Boolean Expressions[80]
Designational Expressions[88]

This macro is invoked in definition 1.

3.1 Variables

3.1.1 Syntax

A `<variable identifier>` in this context is an applied occurrence. We use the symbol `VarIdUse` to play this role, which is quite different from the defining occurrence appearing in Section 5.

Because the phrase structure cannot distinguish different types of expression, we replace `<arithmetic expression>` here with `Expression`. Similarly, `<array identifier>` is replaced with `VarIdUse`.

Variables[54] ≡
```
VarIdUse ::= Identifier .
SimpleVariable ::= VarIdUse .
```

```
SubscriptExpression ::= Expression .
SubscriptList ::=
  SubscriptExpression /
  SubscriptList ',' SubscriptExpression .
SubscriptedVariable ::= VarIdUse '[' SubscriptList ']' .
Variable ::= SimpleVariable / SubscriptedVariable .
```

This macro is invoked in definition 1.

3.1.2 Equivalences

Equivalences[55] ≡
```
Variable ::= SimpleVariable SubscriptedVariable .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

3.1.3 Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements. The type of the value of a particular variable is defined in the declaration for the variable.

`VarIdUse` denotes an identifier that has a type property declared at its defining occurrence, and therefore it plays the `TypedUseId` role in the tree computation that verifies types.

Semantics[56] ≡
```
SYMBOL VarIdUse INHERITS TypedUseId END;
SYMBOL Variable COMPUTE
  SYNT.Sym=CONSTITUENT VarIdUse.Sym SHIELD SubscriptExpression;
  SYNT.Type=CONSTITUENT VarIdUse.Type SHIELD SubscriptExpression;
  SYNT.Key=CONSTITUENT VarIdUse.Key SHIELD SubscriptExpression <- THIS.Type;
  SYNT.kind=GetKind(THIS.Key,UndefinedIdn);
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

The `Kind` property of an identifier is guaranteed to be set if the `Type` property of that variable is set, hence the dependence in the next-to-last line.

3.1.4 Subscripts

Subscripted variables designate values which are components of multidimensional arrays. Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets `[ ]`.

This means that the identifier must denote an array or a switch, and the number of subscripts must equal the number of dimensions.

Subscripts[57] ≡
```
ATTR dim, sub: int;
```

```
RULE: Variable ::= VarIdUse '[' SubscriptList ']' COMPUTE
  .kind=GetKind(VarIdUse.Key,UndefinedIdn) <- VarIdUse.Type;
  IF(AND(AND(NE(.kind,ArrayIdn),NE(.kind,UndefinedIdn)),NE(.kind,SwitchIdn)),
    message(ERROR,"Array or switch identifier required",0,COORDREF));

  .dim=GetDim(VarIdUse.Key,0) <- VarIdUse.Type;
  .sub=
    CONSTITUENTS SubscriptExpression.sub SHIELD SubscriptExpression
      WITH (int, ADD, IDENTICAL, ZERO);
  IF(AND(NE(.dim,0),NE(.dim,.sub)),
      message(ERROR,"Number of indices differs from dimension",0,COORDREF));
END;

RULE: SubscriptExpression ::= Expression COMPUTE
  SubscriptExpression.sub=1;
END;
```

This macro is defined in definitions 57 and 58.
This macro is invoked in definition 7.

Each subscript position acts like a variable of type `integer` and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable. Thus the subscript expression is considered a `RootContext` requiring an integer value and allowing all type conversions defined for assignment.

Subscripts[58] ≡

```
RULE: SubscriptExpression ::= Expression COMPUTE
  RootContext(integerType,,Expression);
  Indication(ColonEqual);
END;
```

This macro is defined in definitions 57 and 58.
This macro is invoked in definition 7.

3.2 Function Designators

3.2.1 Syntax

A `<procedure identifier>` in this context is an applied occurrence. We use the symbol `ProcIdUse` to play this role, which is quite different from the defining occurrence appearing in Section 5.

Function Designators[59] ≡

```
ProcIdUse ::= Identifier .
ActualParameter ::= String / Expression .
ParameterDelimiter ::= ',' / PDLetterString .
```

This macro is defined in definitions 59, 61, and 62.
This macro is invoked in definition 1.

The complex parameter delimiter must be treated as a basic symbol in order to distinguish the letter string it contains from an identifier:

Parameter Delimiter Letter String[60] ≡

```
PDLetterString : $\)[a-zA-Z]+:\(
```

This macro is invoked in definition 2.

Function Designators[61] ≡
```
ActualParameterList ::=
  ActualParameter /
  ActualParameterList ParameterDelimiter ActualParameter .
```

This macro is defined in definitions 59, 61, and 62.
This macro is invoked in definition 1.

A `FunctionDesignator` with an empty `ActualParameterPart` cannot be distinguished from a `Variable` in the absence of type information, so we cannot allow an empty `ActualParameterPart` in this context.

Function Designators[62] ≡
```
ActualParameterPart ::= '(' ActualParameterList ')' .
FunctionDesignator ::= ProcIdUse ActualParameterPart .
```

This macro is defined in definitions 59, 61, and 62.
This macro is invoked in definition 1.

3.2.2 Equivalences

3.2.3 Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration to fixed sets of actual parameters. Thus the `ProcIdUse` denotes an operator that is applied to a list of operands and returns a value.

Semantics[63] ≡
```
SYMBOL FunctionDesignator  INHERITS ExpressionSymbol END;
SYMBOL ProcIdUse           INHERITS OperatorSymbol   COMPUTE
  SYNT.Indic=THIS.Key <- INCLUDING RootType.GotAllOpers;
END;
SYMBOL ActualParameterPart INHERITS OpndExprListRoot END;
SYMBOL ActualParameter     INHERITS OpndExprListElem END;

RULE: ActualParameter ::= String COMPUTE
  PrimaryContext(ActualParameter,stringType);
END;

RULE: ActualParameter ::= Expression COMPUTE
  TransferContext(ActualParameter,Expression);
END;

RULE: FunctionDesignator ::= ProcIdUse ActualParameterPart COMPUTE
  ListContext(FunctionDesignator,ProcIdUse,ActualParameterPart);
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

3.2.4 Standard Functions

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. The report specifically recommends the following, but does not preclude the addition of others:

Standard Functions[64] ≡
```
PreDefKey("abs",   absKey)
PreDefKey("sign",  signKey)
PreDefKey("sqrt",  sqrtKey)
PreDefKey("sin",   sinKey)
PreDefKey("cos",   cosKey)
PreDefKey("arctan",arctanKey)
PreDefKey("ln",    lnKey)
PreDefKey("exp",   expKey)
```
This macro is invoked in definition 66.

`PreDefKey` is exported by an Eli library module.

Instantiate modules to pre-define identifiers[65] ≡
```
$/Name/PreDefine.gnrc +referto=Identifier :inst
$/Name/PreDefId.gnrc +referto=(algol60.d) :inst
```
This macro is invoked in definition 19.

The predefinition module requires that the specifications of predefined identifiers be supplied in a file whose name is passed as the `referto` parameter of the module's instantiation. This file is defined above as the result of the Eli request `ALGOL60.fw:fwGen/algol60.d`:

algol60.d[66] ≡
Standard Functions[64]
Transfer Functions[68]

This macro is attached to a non-product file.

Eli attaches no particular significance to the `d` suffix. The file is also marked as a "non-product" file, which means that Eli will not attempt to process it further in any event. However, it is included in the collection of files available for reference and is therefore accessible via the `referto` parameter of the `inst` derivation.

The standard functions are understood to operate indifferently on arguments both of type `real` and `integer`. They will all yield values of type `real`, except for `sign(E)` which will have values of type `integer`.

Define operators and associate them with indications[67] ≡
```
OPER
  absKey(realType): realType;
  signKey(realType): integerType;
  sqrtKey,
  sinKey,
  cosKey,
  arctanKey,
  lnKey,
  expKey(realType): realType;
INDICATION
  absKey:    absKey;
  signKey:   signKey;
```

```
  sqrtKey:   sqrtKey;
  sinKey:    sinKey;
  cosKey:    cosKey;
  arctanKey: arctanKey;
  lnKey:     lnKey;
  expKey:    expKey;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

All of these operations are defined to apply to `real` operands. In Section 5.1.3 we shall show how an integer value can be accepted wherever the context requires a real value.

3.2.5 Transfer Functions

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely, `entier(E)`, which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not larger than the value of `E`.

Transfer Functions[68] ≡
```
PreDefKey("entier",entierKey)
```

This macro is invoked in definition 66.

Define operators and associate them with indications[69] ≡
```
OPER
  entierKey(realType): integerType;
INDICATION
  entierKey: entierKey;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

3.3 Arithmetic Expressions

3.3.1 Syntax

We have replaced the integer division operator of the reference language with the symbol `div` because of character set limitations. As mentioned earlier, it is impossible to distinguish syntactically among arithmetic, Boolean, and designational expressions. We have therefore chosen to define the `<if clause>` and conditional expression in Section 3.4.

Arithmetic Expressions[70] ≡
```
AddingOperator ::= '+' / '-' .
MultiplyingOperator ::= '*' / '/' / 'div' .
Primary ::=
  UnsignedNumber /
  Variable /
  FunctionDesignator /
  '(' Expression ')' .
Factor ::= Primary / Factor '^' Primary .
```

```
Term ::= Factor / Term MultiplyingOperator Factor .
SimpleArithmeticExpression ::=
  Term /
  '+' Term / '-' Term /
  SimpleArithmeticExpression AddingOperator Term .
```

This macro is defined in definitions 70.
This macro is invoked in definition 53.

The Revised Report makes no syntactic distinction between the monadic and dyadic versions of `+` and `-`. Because this distinction is important for the semantics of the language, we have made it by using literals for the monadic versions.

3.3.2 Equivalences

The syntactic distinction between operators and among expressions is necessary only to resolve operator precedence and association in text with a minimum number of parentheses.

Equivalences[71] ≡

```
Binop ::= AddingOperator MultiplyingOperator .
Expression ::= SimpleArithmeticExpression Term Factor Primary UnsignedNumber .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

3.3.3 Semantics

An arithmetic expression is a rule for computing a numerical value. The type of a number is known syntactically, and the type of a variable is determined from the declaration. Since the type is known directly, each of these is a primary context. As we have seen in Section 3.2, a `FunctionDesignator` plays the `ExpressionSymbol` role and therefore exchanges type information with the `Expression` via a transfer context.

Semantics[72] ≡

```
RULE: Expression ::= UnsignedInteger COMPUTE
  PrimaryContext(Expression,integerType);
END;

RULE: Expression ::= UnsignedReal COMPUTE
  PrimaryContext(Expression,realType);
END;

RULE: Expression ::= Variable COMPUTE
  PrimaryContext(Expression,Variable.Type);
END;

RULE: Expression ::= FunctionDesignator COMPUTE
  TransferContext(Expression, FunctionDesignator);
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

3.3.4 Operators and Types

The constituents of simple arithmetic expressions must be of types `real` or `integer`. The meanings of the basic operators and the types of the expressions to which they lead are given by the rules in this section.

The operators `+`, `-` and `*` have the conventional meaning (addition, subtraction and multiplication). The type of the expression will be `integer` if both of the operands are of `integer` type, otherwise `real`:

Establish operator indications for source symbols[73] ≡
    Set a monadic operator indication[11](‘`+`’,‘`Nop`’)
    Set a dyadic operator indication[12](‘`+`’,‘`Plus`’)
    Set a monadic operator indication[11](‘`-`’,‘`Neg`’)
    Set a dyadic operator indication[12](‘`-`’,‘`Minus`’)
    Set a dyadic operator indication[12](‘`*`’,‘`Star`’)

This macro is defined in definitions 73, 75, 77, 84, and 86.
This macro is invoked in definition 7.

Define operators and associate them with indications[74] ≡

```
OPER
  iNop,  iNeg(integerType):                                  integerType;
  iiAdd, iiSubtract, iiMultiply(integerType, integerType): integerType;
  rNop,  rNeg(realType):                                     realType;
  rrAdd, rrSubtract, rrMultiply(realType, realType):         realType;
INDICATION
  Nop:   iNop, rNop;
  Plus:  iiAdd, rrAdd;
  Neg:   iNeg, rNeg;
  Minus: iiSubtract, rrSubtract;
  Star:  iiMultiply, rrMultiply;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

It is unnecessary to provide operators for all combinations of integer and real operands because Section 5.1.3 implies that the compiler is allowed to convert an integer value to a real value in an arithmetic expression wherever the context requires it.

The source language operator symbols `/` and `div` both denote division, to be understood as a multiplication of the left operand by the reciprocal of the right operand.

`/` is defined for all four combinations of types `real` and `integer` and will yield results of type `real` in any case. The operator `div` is defined only for two operands both of type `integer` and will yield a result of type `integer`:

Establish operator indications for source symbols[75] ≡
    Set a dyadic operator indication[12](‘`/`’,‘`Slash`’)
    Set a dyadic operator indication[12](‘`div`’,‘`Div`’)

This macro is defined in definitions 73, 75, 77, 84, and 86.
This macro is invoked in definition 7.

Define operators and associate them with indications[76] ≡

```
OPER
  rrDiv(realType, realType):       realType;
  iiDiv(integerType, integerType): integerType;
```

```
INDICATION
  Slash: rrDiv;
  Div:   iiDiv;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

The source language operator symbols `^` denotes exponentiation:

Establish operator indications for source symbols[77] ≡
    Set a dyadic operator indication[12](‘`^`’,‘`UpArrow`’)

This macro is defined in definitions 73, 75, 77, 84, and 86.
This macro is invoked in definition 7.

The Report specifies that the result of raising an integer value to an integer power has type `integer` if the exponent is positive and has type `real` if the exponent is negative. The fact that the *type* of the result of an exponentiation depends on the *value* of one of the operands is a problem, because type is considered to be a static property of the program in this specification. We must therefore define a new type, `arithType` to handle this situation.

Define operators and associate them with indications[78] ≡

```
OPER
  iiExp(integerType, integerType): arithType;
  riExp(realType,    integerType): realType;
  rrExp(realType,       realType): realType;
INDICATION
  UpArrow: iiExp, riExp, rrExp;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

A value of `arithType` type would have to be implemented by a structure consisting of either an integer value or a real value and a flag specifying the type of that value. When using a value of `arithType` type as an operand, a run-time check must be made to determine the actual type and select the appropriate code to carry out the specified operation.

Applying a `Uniop` to a value of `arithType` type, or applying a `Binop` to two values of `arithType` type, will always yield a value of `arithType` type as a result. In other cases, the result may be guaranteed to be of type `real`. A value of `arithType` type can always be converted into a value of type `real` if the context requires it. If the context requires an integer value, then either a value of `arithType` type can be converted to an integer or a run-time error can be reported. All of the possibilities are enumerated here:

Define operators and associate them with indications[79] ≡

```
OPER
  aNop, aNeg(arithType):                                          arithType;
  aaAdd, aaSubtract, aaMultiply, aaExp(arithType, arithType):     arithType;
  aiAdd, aiSubtract, aiMultiply, aiExp(arithType, integerType):   arithType;
  iaAdd, iaSubtract, iaMultiply, iaExp(integerType, arithType):   arithType;
  arAdd, arSubtract, arMultiply, arExp(arithType, realType):      realType;
  raAdd, raSubtract, raMultiply, raExp(realType, arithType):      realType;
INDICATION
  Nop:     aNop;
  Plus:    aaAdd, aiAdd, iaAdd, arAdd, raAdd;
  Neg:     aNeg;
  Minus:   aaSubtract, aiSubtract, iaSubtract, arSubtract, raSubtract;
```

```
  Star:    aaMultiply, aiMultiply, iaMultiply, arMultiply, raMultiply;
  UpArrow: aaExp, aiExp, iaExp, arExp, raExp;
COERCION
  aiConvert(arithType): integerType;
  arConvert(arithType): realType;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

3.4 Boolean Expressions

3.4.1 Syntax

As noted earlier, it is impossible to decide syntactically whether a variable should be a primary or a Boolean primary. If we define a logical value as a Boolean primary, this introduces an asymmetry in error reporting: use of a logical value in an arithmetic context is a syntax error, but use of a Boolean variable in the same context is a type error. This asymmetry is avoided by defining a logical value as a primary rather than a Boolean primary.

We used obvious representations for the operators that are not available as ASCII characters:

Boolean Expressions[80] ≡
```
Primary ::= LogicalValue .
RelationalOperator ::= '<' / '<=' / '=' / '>=' / '>' / '!=' .
Relation ::=
  SimpleArithmeticExpression RelationalOperator
  SimpleArithmeticExpression .
BooleanPrimary ::=
  SimpleArithmeticExpression /
  Relation .
BooleanSecondary ::= BooleanPrimary / 'not' BooleanPrimary .
BooleanFactor ::= BooleanSecondary /
                  BooleanFactor 'and' BooleanSecondary .
BooleanTerm ::= BooleanFactor / BooleanTerm 'or' BooleanFactor .
Implication ::= BooleanTerm / Implication '->' BooleanTerm .
SimpleBoolean ::= Implication / SimpleBoolean '==' Implication .
IfClause ::= 'if' Expression 'then' .
Expression ::=
  SimpleBoolean /
  IfClause SimpleBoolean 'else' Expression .
```

This macro is invoked in definition 53.

3.4.2 Equivalences

Equivalences[81] ≡
```
Binop ::= RelationalOperator .
Expression ::=
  SimpleBoolean Implication BooleanTerm BooleanFactor BooleanSecondary
  BooleanPrimary Relation LogicalValue.
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

3.4.3 Semantics

A Boolean expression is a rule for computing a logical value.

Semantics[82] ≡
```
RULE: Expression ::= 'true' COMPUTE
  PrimaryContext(Expression,BooleanType);
END;

RULE: Expression ::= 'false' COMPUTE
  PrimaryContext(Expression,BooleanType);
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

3.4.4 Types

Variables and function designators entered as Boolean primaries must be declared `Boolean`. The expression typing is handled by the rules given in Section 3.3.3.

The `Expression` in an `IfClause` must yield a Boolean value.

Semantics[83] ≡
```
RULE: IfClause ::= 'if' Expression 'then' COMPUTE
  Expression.Required=BooleanType;
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

3.4.5 The Operators

Relations take on the value `true` whenever the corresponding relation is satisfied for the expressions involved, otherwise `false`.

Establish operator indications for source symbols[84] ≡
Set a monadic operator indication[11](‘`not`’,‘`Not`’)
Set a dyadic operator indication[12](‘`<`’,‘`Lt`’)
Set a dyadic operator indication[12](‘`<=`’,‘`Le`’)
Set a dyadic operator indication[12](‘`=`’,‘`Eq`’)
Set a dyadic operator indication[12](‘`!=`’,‘`Ne`’)
Set a dyadic operator indication[12](‘`>=`’,‘`Ge`’)
Set a dyadic operator indication[12](‘`>`’,‘`Gt`’)

This macro is defined in definitions 73, 75, 77, 84, and 86.
This macro is invoked in definition 7.

Define operators and associate them with indications[85] ≡
```
OPER
  iiLT, iiLE, iiEQ, iiGE, iiGT, iiNE(integerType, integerType): BooleanType;
  rrLT, rrLE, rrEQ, rrGE, rrGT, rrNE(realType, realType):       BooleanType;
INDICATION
```

```
  Lt: iiLT, rrLT;
  Le: iiLE, rrLE;
  Eq: iiEQ, rrEQ;
  Ge: iiGE, rrGE;
  Gt: iiGT, rrGT;
  Ne: iiNE, rrNE;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

The logical operators always take values of type `Boolean` as operands and yield values of type `Boolean` as results.

Establish operator indications for source symbols[86] ≡
Set a dyadic operator indication[12](‘`and`’,‘`And`’)
Set a dyadic operator indication[12](‘`or`’,‘`Or`’)
Set a dyadic operator indication[12](‘`->`’,‘`Implies`’)
Set a dyadic operator indication[12](‘`==`’,‘`Equiv`’)

This macro is defined in definitions 73, 75, 77, 84, and 86.
This macro is invoked in definition 7.

Define operators and associate them with indications[87] ≡

```
OPER
  bNot(BooleanType): BooleanType;
  bAnd, bOr, bImplies, bEquiv(BooleanType, BooleanType): BooleanType;
INDICATION
  Not:     bNot;
  And:     bAnd;
  Or:      bOr;
  Implies: bImplies;
  Equiv:   bEquiv;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

3.5 Designational Expressions

3.5.1 Syntax

We do not allow unsigned integer labels because of an ambiguity that arises when a designational expression is used as an actual parameter. (This ambiguity was reported in ALGOL Bulletin No. 10, October, 1960.)

Designational Expressions[88] ≡

```
Label ::= Identifier .
```

This macro is invoked in definition 53.

Designational expressions do not appear in the syntax here because their phrase structure is already covered by other forms of expression, and they cannot be distinguished without type information.

3.5.2 Equivalences

3.5.3 Semantics

A designational expression is a rule for obtaining a label of a statement.

Semantics[89] ≡
```
RULE: DesignationalExpression ::= Expression COMPUTE
  Expression.Required=labelType;
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

4 Statements

4.1 Compound Statements and Blocks

4.1.1 Syntax

Compound Statements and Blocks[90] ≡
```
UnlabelledBasicStatement ::=
  AssignmentStatement /
  GoToStatement /
  DummyStatement /
  ProcedureStatement .
BasicStatement ::= UnlabelledBasicStatement / Label ':' BasicStatement.
UnconditionalStatement ::= BasicStatement / CompoundStatement / Block .
Statement ::= UnconditionalStatement / ConditionalStatement / ForStatement .
CompoundTail ::= Statement 'end' / Statement ';' CompoundTail .
BlockHead ::= 'begin' Declaration / BlockHead ';' Declaration .
UnlabelledCompound ::= 'begin' CompoundTail .
UnlabelledBlock ::= BlockHead ';' CompoundTail .
CompoundStatement ::= UnlabelledCompound / Label ':' CompoundStatement .
Block ::= UnlabelledBlock / Label ':' Block .
Program ::= Block / CompoundStatement .
```

This macro is invoked in definition 1.

4.1.2 Equivalences

Equivalences[91] ≡
```
Statement ::=
  UnlabelledBasicStatement BasicStatement UnlabelledCompound CompoundStatement
  UnconditionalStatement .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

4.1.3 Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration be specified to be local to the block in question. The behavior described in detail in the Revised Report is provided by the `RangeScope` computational role, exported by the name analysis module.

Semantics[92] ≡
```
SYMBOL UnlabelledBlock INHERITS RangeScope END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

4.2 Assignment Statements

4.2.1 Syntax

The nonterminals `<arithmetic expression>` and `<Boolean expression>` must be replaced by `Expression`, in accordance with the changes made in Section 3. Also, as discussed in Section 3, distinctions among identifiers are often dropped. Therefore there is only a single rule for `LeftPart` and a single rule for `AssignmentStatement`:

Assignment Statements[93] ≡
```
LeftPart ::= Variable ':=' .
LeftPartList ::= LeftPart / LeftPartList LeftPart .
AssignmentStatement ::= LeftPartList Expression .
```

This macro is invoked in definition 1.

4.2.2 Equivalences

Equivalences[94] ≡
```
Statement ::= AssignmentStatement .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

4.2.3 Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers.

Semantics[95] ≡
```
RULE: LeftPart ::= Variable ':=' COMPUTE
  IF(AND(AND(AND(
      NE(Variable.kind,VariableIdn),
      NE(Variable.kind,ArrayIdn)),
      NE(Variable.kind,ProcedureIdn)),
      NE(Variable.kind,UndefinedIdn)),
    message(ERROR,"Illegal target of assignment :=",0,COORDREF));
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

Assignments to a procedure identifier may only occur within the body of a procedure defining the value of a function designator. This is checked by defining an `InBody` property for each procedure identifier. `InBody` has the value 0 outside of the procedure body and a nonzero value within the body. The inBody chain is used to control the value of `InBody`.

Semantics[96] ≡
```
RULE: LeftPart ::= Variable ':=' COMPUTE
  IF(AND(
      EQ(Variable.kind,ProcedureIdn),
      EQ(GetInBody(Variable.Key,0),0)<-LeftPart.inBody),
    message(
      ERROR,
      CatStrInd(
        "Assignment must be within the body of procedure ",
        Variable.Sym),
      0,
      COORDREF));
END;
```
This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

4.2.4 Types

The type associated with all variables and procedure identifiers of a left part must be the same.

Verify type consistency of a left-part list[97] ≡
```
RULE: LeftPartList ::= LeftPart COMPUTE
  LeftPartList.Type=LeftPart.Type;
END;

RULE: LeftPartList ::= LeftPartList LeftPart COMPUTE
  LeftPartList[1].Type=LeftPart.Type;
  IF(NE(LeftPartList[2].Type,LeftPart.Type),
    message(ERROR,"Type does not agree with previous variable",0,COORDREF));
END;

RULE: LeftPart ::= Variable ':=' COMPUTE
  LeftPart.Type=Variable.Type;
END;
```
This macro is invoked in definition 7.

If this type is `Boolean`, the expression must likewise be `Boolean`. If the type is `real` or `integer`, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked.

Verify that an expression can be assigned to a left-part list[98] ≡
```
RULE: Statement ::= LeftPartList Expression COMPUTE
```

```
    RootContext(LeftPartList.Type,,Expression);
    Indication(ColonEqual);
  END;
```

This macro is invoked in definition 7.

For transfer from `real` to `integer` type, the transfer function is understood to yield a result equivalent to `entier(E+0.5)` where `E` is the value of the expression.

Define operators and associate them with indications[99] ≡

```
  OPER
    riRound(realType): integerType;
  INDICATION
    ColonEqual: riRound;
```

This macro is defined in definitions 67, 69, 74, 76, 78, 79, 85, 87, and 99.
This macro is invoked in definition 13.

4.3 Go To Statements

4.3.1 Syntax

Go To Statements[100] ≡

```
  GoToStatement ::= 'go' 'to' DesignationalExpression .
```

This macro is invoked in definition 1.

4.3.2 Equivalences

Equivalences[101] ≡

```
  Statement ::= GoToStatement .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

4.3.3 Semantics

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly be the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

The `jump` operator is never used. It serves merely to establish the `labelType`.

Define operators to establish types without values[102] ≡

```
  OPER
    jump(labelType):voidType;
```

This macro is defined in definitions 102 and 119.
This macro is invoked in definition 13.

4.4 Dummy Statements

Dummy Statements[103] ≡
```
DummyStatement ::= /*Empty*/ .
```

This macro is invoked in definition 1.

4.4.1 Equivalences

Equivalences[104] ≡
```
Statement ::= DummyStatement .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

4.4.2 Semantics

A dummy statement executes no operation. It may serve to place a label.

4.5 Conditional Statements

We have deleted the definition of `IfClause`, which duplicates the one in Section 3, and the definition of `UnconditionalStatement`, which duplicates the one in Section 4.1.

Conditional Statements[105] ≡
```
IfStatement ::= IfClause UnconditionalStatement .
ConditionalStatement ::=
  IfStatement /
  IfStatement 'else' Statement /
  IfClause ForStatement /
  Label ':' ConditionalStatement .
```

This macro is invoked in definition 1.

4.5.1 Equivalences

Equivalences[106] ≡
```
Statement ::= ConditionalStatement IfStatement .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

4.5.2 Semantics

No tree computations are required here. The type of the expression in the `IfClause` is verified by the rule in Section 3.4.4.

4.6 For Statements

4.6.1 Syntax

`ArithmeticExpression` and `BooleanExpression` can be recognized here on the basis of context.

For Statements[107] ≡
```
ArithmeticExpression ::= Expression .
BooleanExpression ::= Expression .
ForListElement ::=
  ArithmeticExpression /
  ArithmeticExpression 'step' ArithmeticExpression 'until' ArithmeticExpression /
  ArithmeticExpression 'while' BooleanExpression .
ForList ::= ForListElement / ForList ',' ForListElement .
ForClause ::= 'for' Variable ':=' ForList 'do' .
ForStatement ::= ForClause Statement / Label ':' ForStatement .
```

This macro is invoked in definition 1.

4.6.2 Equivalences

Equivalences[108] ≡
```
Statement ::= ForStatement .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

4.6.3 Semantics

A for clause causes the statement S which it precedes to be executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The syntax of the report constrains the controlled variable to be of arithmetic type.

Semantics[109] ≡
```
RULE: ForClause ::= 'for' Variable ':=' ForList 'do' COMPUTE
  ForClause.Type=Variable.Type;
  IF(AND(
      NE(Variable.Type,integerType),
      NE(Variable.Type,realType)),
    message(ERROR,"Controlled variable must be arithmetic",0,COORDREF));
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

4.6.4 The for List Elements

The for list gives a rule for obtaining values that are consecutively assigned to the controlled variable. The value of each `ArithmeticExpression` must be assigned to the controlled variable.

The for List Elements[110] ≡

```
RULE: ArithmeticExpression ::= Expression COMPUTE
  RootContext(INCLUDING ForClause.Type,,Expression);
  Indication(ColonEqual);
END;

RULE: BooleanExpression ::= Expression COMPUTE
  Expression.Required=BooleanType;
END;
```

This macro is invoked in definition 7.

4.7 Procedure Statements

4.7.1 Syntax

We have deleted the definitions of `<actual parameter>`, `<parameter delimiter>`, `<actual parameter list>` and `<actual parameter part>`. All of these nonterminals are defined in Section 3.2. (See the discussion there concerning the ambiguity of a letter string that is part of a parameter delimiter.) In addition, `<procedure identifier>` is replaced by `ProcIdUse` in order to capture the role of the identifier as an applied occurrence.

The nonterminal `NoArgs` is used to represent an empty argument list.

Procedure Statements[111] ≡
```
ProcedureStatement ::= ProcIdUse NoArgs / ProcIdUse ActualParameterPart .
NoArgs ::= .
```

This macro is invoked in definition 1.

4.7.2 Equivalences

`NoArgs` is sematically equivalent to an argument list. Using this equivalence simplifies the process of checking the argument list against the formal parameters of the procedure invoked.

Equivalences[112] ≡
```
ActualParameterPart ::= NoArgs .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

4.7.3 Semantics

A procedure used in a procedure statement cannot return a value.

Semantics[113] ≡
```
SYMBOL ProcedureStatement INHERITS ExpressionSymbol COMPUTE
  INH.Required=voidType;
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created with the formal parameters declared as variables local to this fictitious block.

Semantics[114] ≡
```
SYMBOL ValueParameter COMPUTE
  SYNT.GotValue=ResetIsValue(THIS.Key, 1);
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

Properties characterizing formal parameters[115] ≡
```
IsValue:  int;
```

This macro is invoked in definition 16.

4.7.4 Actual-Formal Correspondence

The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order. Use of the correspondence is made when checking the restrictions stated in Section 4.7.5.

Actual-Formal Correspondence[116] ≡
```
SYMBOL ActualParameterPart INHERITS ParameterDeListRoot END;
SYMBOL ActualParameter     INHERITS ParameterDeListElem END;

RULE: FunctionDesignator ::= ProcIdUse ActualParameterPart COMPUTE
  ActualParameterPart.ParameterList=
    GetFormals(ProcIdUse.Key,NULLParameterList)
    <- INCLUDING RootType.GotAllOpers;
END;

RULE: ProcedureStatement ::= ProcIdUse ActualParameterPart COMPUTE
  ActualParameterPart.ParameterList=
    GetFormals(ProcIdUse.Key,NULLParameterList)
    <- INCLUDING RootType.GotAllOpers;
END;
```

This macro is defined in definitions 116.
This macro is invoked in definition 7.

Instantiate modules for list handling[117] ≡
```
$/Adt/LidoList.gnrc +instance=Parameter +referto=algol60 :inst
```

This macro is invoked in definition 19.

A parameter in this list is really a definition table key, although we could make it a more complex value if necessary.

Parameter identification[118] ≡
```
#include "deftbl.h"
```

```
#define NoParameter NoKey
typedef DefTableKey Parameter;
```

This macro is invoked in definition 17.

The actual verification that the procedure is called with the right number of arguments of the correct type is carried out during type analysis.

4.7.5 Restrictions

If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code) then this string can only be used in the procedure body as an actual parameter for further calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code. Thus there is no operator that takes a string as an argument or delivers one as a result, and we must create one in order to establish the `stringType`.

Define operators to establish types without values[119] ≡

```
OPER
  pass(stringType):voidType;
```

This macro is defined in definitions 102 and 119.
This macro is invoked in definition 13.

A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable.

Restrictions[120] ≡

```
ATTR IsAssigned, IsVariable: int;

SYMBOL             Variable      COMPUTE      INH.IsAssigned=0; END;
RULE: LeftPart ::= Variable ':=' COMPUTE Variable.IsAssigned=1; END;

SYMBOL Expression              COMPUTE       SYNT.IsVariable=0; END;
RULE:  Expression ::= Variable COMPUTE Expression.IsVariable=1; END;

SYMBOL VarIdUse COMPUTE
  SYNT.GotAssign=ResetIsAssigned(THIS.Key,INCLUDING Variable.IsAssigned);
END;

SYMBOL ROOTCLASS COMPUTE
  SYNT.GotProperties=CONSTITUENTS (VarIdUse.GotAssign,ValueParameter.GotValue);
END;

RULE: ActualParameter ::= Expression COMPUTE
  IF(AND(AND(
      GetIsAssigned(ActualParameter.ParameterElem,0),
      NOT(GetIsValue(ActualParameter.ParameterElem,0))),
      NOT(Expression.IsVariable)),
    message(ERROR,"Argument must be a variable",0,COORDREF))
  <- INCLUDING (ROOTCLASS.GotProperties);
END;
```

This macro is defined in definitions 120, 122, 123, and 124.
This macro is invoked in definition 7.

Property to support Section 4.7.5.2 check[121] ≡

```
IsAssigned: int;
```

This macro is invoked in definition 16.

A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array.

Restrictions[122] ≡

```
SYMBOL Expression             COMPUTE       SYNT.kind=UndefinedIdn;  END;
RULE: Expression ::= Variable COMPUTE Expression.kind=Variable.kind; END;

RULE: ActualParameter ::= Expression COMPUTE
  IF(AND(
      EQ(GetKind(ActualParameter.ParameterElem,UndefinedIdn),ArrayIdn),
      NE(Expression.kind,ArrayIdn)),
    message(ERROR,"Argument must be an array",0,COORDREF));
END;
```

This macro is defined in definitions 120, 122, 123, and 124.
This macro is invoked in definition 7.

A formal parameter which is called by value cannot in general correspond to a switch identifier or a string, because these do not possess values.

Restrictions[123] ≡

```
SYMBOL ValueParameter INHERITS TypedUseId COMPUTE
  IF(EQ(GetKind(THIS.Key,UndefinedIdn),SwitchIdn) <- THIS.Type,
    message(
      ERROR,
      CatStrInd("Value parameter cannot be a switch identifier: ",THIS.Sym),
      0,
      COORDREF));
  IF(EQ(THIS.Type,stringType),
    message(
      ERROR,
      CatStrInd("Value parameter cannot be a string: ",THIS.Sym),
      0,
      COORDREF));
END;
```

This macro is defined in definitions 120, 122, 123, and 124.
This macro is invoked in definition 7.

All formal parameters have restrictions on the type of the corresponding actual parameter. (This is a change from the Revised Report, discussed in Section 1.)

Restrictions[124] ≡

```
RULE: ProcedureStatement ::= ProcIdUse ActualParameterPart COMPUTE
  ListContext(ProcedureStatement,ProcIdUse,ActualParameterPart);
END;
```

This macro is defined in definitions 120, 122, 123, and 124.
This macro is invoked in definition 7.

5 Declarations

Declarations serve to define certain properties of the quantities used in the program, and associate them with identifiers. A declaration of an identifier is valid for one block. Outside the block the particular identifier may be used for other purposes.

Declarations[125] ≡
```
Declaration ::=
  TypeDeclaration /
  ArrayDeclaration /
  SwitchDeclaration /
  ProcedureDeclaration .
```

Type Declarations[128]
Array Declarations[132]
Switch Declarations[140]
Procedure Declarations[142]

This macro is invoked in definition 1.

Dynamically this implies the following: at the time of an entry to the block (through the `begin`, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from the block (through `end` or by a go to statement) all identifiers which are declared for the block lose their local significance.

Eli captures this behavior in the `IdUseEnv` computational role of the name analysis module. The `ChkIdUse` computational role emits an error report if the identifier has not been declared.

Scope Violation Rules[126] ≡
```
SYMBOL VarIdUse  INHERITS IdentOcc, IdUseEnv, ChkIdUse END;
SYMBOL ProcIdUse INHERITS IdentOcc, IdUseEnv, ChkIdUse END;
```

This macro is defined in definitions 37, 126, and 127.
This macro is invoked in definition 7.

Labels are implicitly declared by their appearance preceding a colon. All other identifiers of a program must be explicitly declared. No identifier may be declared more than once in any block head.

The `IdDefScope` computational role of the name analysis module provides for identifier declaration. `MultDefChk` was defined in Section 2.4 to report any identifier that is defined more than once in a block head.

Scope Violation Rules[127] ≡
```
SYMBOL VarIdDef            INHERITS IdentOcc, IdDefScope, MultDefChk END;
SYMBOL ArrayIdentifier     INHERITS IdentOcc, IdDefScope, MultDefChk END;
SYMBOL ProcedureIdentifier INHERITS IdentOcc, IdDefScope, MultDefChk END;
SYMBOL Label               INHERITS IdentOcc, IdDefScope, MultDefChk END;
SYMBOL SwitchIdentifier    INHERITS IdentOcc, IdDefScope, MultDefChk END;
```

This macro is defined in definitions 37, 126, and 127.
This macro is invoked in definition 7.

5.1 Type Declarations

5.1.1 Syntax

The Revised Report defines the elements of a `TypeList` to be `SimpleVariable`s. This definition does not reflect the difference in semantics between definition and use. In order to expose that difference, we have used the symbol `VarIdDef` here. The effect of this definition is identical to that of the report, which defines a `<simple variable>` as a `<variable identifier>` in Section 3.1.1.

Type Declarations[128] ≡
```
VarIdDef ::= Identifier .
TypeList ::= VarIdDef / VarIdDef ',' TypeList .
Type ::= 'real' / 'integer' / 'Boolean' .
LocalOrOwnType ::= Type / 'own' Type .
TypeDeclaration ::= LocalOrOwnType TypeList .
```

This macro is invoked in definition 125.

5.1.2 Equivalences

Equivalences[129] ≡
```
Type ::= LocalOrOwnType .
```

This macro is defined in definitions 55, 71, 81, 91, 94, 101, 104, 106, 108, 112, and 129.
This macro is invoked in definition 6.

5.1.3 Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. The Eli `Typing` module's `TypedDefinition` role is played by the `TypeDeclaration`, the symbol that represents a definition of one or more simple variables. Each of those variables plays the `TypedDefId` role.

The `Type` attribute of the symbol playing the `TypedDefinition` role must be set by a user computation to the definition table key of the type. No other user computations are needed.

Semantics[130] ≡
```
SYMBOL TypeDeclaration INHERITS TypedDefinition END;
SYMBOL VarIdDef        INHERITS TypedDefId      COMPUTE
  INH.GotProp=ResetKind(THIS.Key,VariableIdn);
END;

RULE: Type ::= 'real'    COMPUTE Type.Type=realType;    END;
RULE: Type ::= 'integer' COMPUTE Type.Type=integerType; END;
RULE: Type ::= 'Boolean' COMPUTE Type.Type=BooleanType; END;

RULE: TypeDeclaration ::= Type TypeList COMPUTE
  TypeDeclaration.Type=Type.Type;
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

Define an operator to be applied when needed[131] ≡
```
COERCION
  irConvert(integerType): realType;
```
This macro is defined in definitions 131.
This macro is invoked in definition 13.

5.2 Array Declarations

5.2.1 Syntax

The nonterminal `<arithmetic expression>` is replaced by `Expression` and array identifiers are not distinguished, as discussed in Section 3.

Array Declarations[132] ≡
```
LowerBound ::= Expression .
UpperBound ::= Expression .
BoundPair ::= LowerBound ':' UpperBound .
BoundPairList ::= BoundPair / BoundPairList ',' BoundPair .
ArrayIdentifier ::= Identifier .
ArraySegment ::=
  ArrayIdentifier '[' BoundPairList ']' / ArrayIdentifier ',' ArraySegment .
ArrayList ::= ArraySegment / ArrayList ',' ArraySegment .
ArrayDeclaration ::= 'array' ArrayList / LocalOrOwnType 'array' ArrayList .
```
This macro is invoked in definition 125.

5.2.2 Equivalences

5.2.3 Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts, and the types of the variables.

All arrays declared in one declaration are of the same quoted type. If not type declarator is given the type `real` is understood.

The Eli `Typing` module's `TypedDefinition` role is played by the `ArrayDeclaration`, the symbol that represents a definition of one or more subscripted variables. Each of those variables plays the `TypedDefId` role.

Semantics[133] ≡
```
SYMBOL ArrayDeclaration INHERITS TypedDefinition END;
SYMBOL ArrayIdentifier  INHERITS TypedDefId      END;

RULE: ArrayDeclaration ::= Type 'array' ArrayList COMPUTE
  ArrayDeclaration.Type=Type.Type;
END;
```

```
RULE: ArrayDeclaration ::= 'array' ArrayList COMPUTE
  ArrayDeclaration.Type=realType;
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

The dimensions are given as the number of entries in the bound pair list.

Semantics[134] ≡
```
SYMBOL BoundPair COMPUTE SYNT.dim=1; END;

RULE: ArraySegment ::= ArrayIdentifier '[' BoundPairList ']' COMPUTE
  ArraySegment.dim=
    CONSTITUENTS BoundPair.dim WITH (int, ADD, IDENTICAL, ZERO);
  ArrayIdentifier.dim=ArraySegment.dim;
END;

RULE: ArraySegment ::= ArrayIdentifier ',' ArraySegment COMPUTE
  ArraySegment[1].dim=ArraySegment[2].dim;
  ArrayIdentifier.dim=ArraySegment[2].dim;
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

The declared dimensions are stored as the value of the `Dim` property of the `ArrayIdentifier`.

Dimension property of an array[135] ≡
```
Dim: int;
```

This macro is invoked in definition 16.

The `Dim` and `Kind` properties of the variable must be available whenever the type is available, which is guaranteed by making `ArrayIdentifier.GotProp` dependent upon setting them.

Semantics[136] ≡
```
SYMBOL ArrayIdentifier COMPUTE
  INH.GotProp=
    ORDER(
      ResetDim(THIS.Key,THIS.dim),
      ResetKind(THIS.Key,ArrayIdn));
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

5.2.4 Lower upper bound expressions

Bound expressions will be evaluated in the same way as subscript expressions (Section 3.1.4)

Lower upper bound expressions[137] ≡
```
RULE: LowerBound ::= Expression COMPUTE
  RootContext(integerType,,Expression);
  Indication(ColonEqual);
```

```
END;

RULE: UpperBound ::= Expression COMPUTE
  RootContext(integerType,,Expression);
  Indication(ColonEqual);
END;
```

This macro is defined in definitions 137 and 138.
This macro is invoked in definition 7.

Bound expressions can only depend on variables and procedures which are nonlocal to the block for which the array declaration is valid.

Lower upper bound expressions[138] ≡

```
ATTR InBoundExpr: int;

SYMBOL ROOTCLASS COMPUTE SYNT.InBoundExpr=0; END;
SYMBOL BoundPair COMPUTE SYNT.InBoundExpr=1; END;

SYMBOL AnyScope COMPUTE SYNT.Key=NewKey(); END;
SYMBOL Quantity COMPUTE
  SYNT.GotBlk=ResetBlock(THIS.Key,INCLUDING AnyScope.Key);
END;

SYMBOL VarIdDef             INHERITS Quantity END;
SYMBOL ArrayIdentifier      INHERITS Quantity END;
SYMBOL ProcedureIdentifier INHERITS Quantity END;

SYMBOL ROOTCLASS COMPUTE
  SYNT.GotBlks=CONSTITUENTS Quantity.GotBlk;
END;

SYMBOL VarIdUse COMPUTE
  IF(AND(AND(
      NE(GetKind(THIS.Key,UndefinedIdn),UndefinedIdn),
      INCLUDING (ROOTCLASS.InBoundExpr, BoundPair.InBoundExpr)),
      EQ(GetBlock(THIS.Key,NoKey),INCLUDING AnyScope.Key)),
    message(
      ERROR,
      CatStrInd("Local quantity in a bound expression: ",THIS.Sym),
      0,
      COORDREF));
END;
```

This macro is defined in definitions 137 and 138.
This macro is invoked in definition 7.

Property to support Section 5.2.4.2 check[139] ≡

```
Block: DefTableKey;
```

This macro is invoked in definition 16.

5.3 Switch Declarations

5.3.1 Syntax

The nonterminal `<designational expression>` is replaced by `Expression`, as discussed in Section 3.

Switch Declarations[140] ≡
```
SwitchList ::= DesignationalExpression / SwitchList ',' DesignationalExpression .
SwitchIdentifier ::= Identifier .
SwitchDeclaration ::= 'switch' SwitchIdentifier ':=' SwitchList .
```

This macro is invoked in definition 125.

5.3.2 Equivalences

5.3.3 Semantics

A switch declaration defines the set of values of the corresponding switch designators. These values are always labels, and the result is therefore a label.

Semantics[141] ≡
```
SYMBOL SwitchDeclaration INHERITS TypedDefinition COMPUTE
  SYNT.Type=labelType;
END;

SYMBOL SwitchIdentifier INHERITS TypedDefId COMPUTE
  INH.GotProp=
    ORDER(
      ResetKind(THIS.Key,SwitchIdn),
      ResetDim(THIS.Key,1));
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

5.4 Procedure Declarations

5.4.1 Syntax

If there are no formal parameters, the `ValuePart` and the `SpecificationPart` must also be empty. This is only one of several constraints on identifiers appearing in a procedure heading that cannot be checked syntactically. In order to support semantic checking, we need to distinguish the various roles that identifiers play in procedure definition. For example, we use the symbol `ValueParameter` to characterize an identifier appearing in the `ValuePart`.

Procedure Declarations[142] ≡
```
FormalParameter ::= Identifier .
FormalParameterList ::=
  FormalParameter /
  FormalParameterList ParameterDelimiter FormalParameter .
FormalParameterPart ::= / '(' FormalParameterList ')' .
```

```
IdentifierList ::= ValueParameter / IdentifierList ',' ValueParameter .
ValueParameter ::= Identifier .
ValuePart ::= 'value' IdentifierList ';' / /*Empty*/ .
```

This macro is defined in definitions 142, 143, and 144.
This macro is invoked in definition 125.

The syntax of `SpecificationPart` in the report is ambiguous. Also, we need to distinguish the role played by the identifiers in the specification part: they constitute the type definitions of the formal parameters.

Procedure Declarations[143] ≡

```
Specifier ::=
  'string' /
  Type /
  'array' /
  Type 'array' /
  'label' /
  'switch' /
  'procedure' /
  Type 'procedure' .
SpecificationPart ::= FormalSpecification* .
FormalSpecification ::= Specifier FormalIdList ';' .
FormalIdList ::= FormalParamIdTypeDef / FormalIdList ',' FormalParamIdTypeDef .
FormalParamIdTypeDef ::= Identifier .
```

This macro is defined in definitions 142, 143, and 144.
This macro is invoked in definition 125.

The grammar given in the Revised Report does not reflect the scope rules of the language: A procedure identifier is declared in the enclosing scope, and the formal parameters are declared in the procedure body, but the grammar places both in the `<procedure heading>` phrase. We have therefore moved the procedure identifier to the procedure declaration and defined a new nonterminal, `ProcedureRange`, that is exactly the scope of the formal parameters described by Section 5.4.3 of the Revised Report:

Procedure Declarations[144] ≡

```
ProcedureIdentifier ::= Identifier .
ProcedureHeading ::= FormalParameterPart ';' ValuePart SpecificationPart .
ProcedureBody ::= Statement / Code .
ProcedureRange ::=
  ProcedureHeading ProcedureBody .
ProcedureDeclaration ::=
  'procedure' ProcedureIdentifier ProcedureRange /
  Type 'procedure' ProcedureIdentifier ProcedureRange .
```

This macro is defined in definitions 142, 143, and 144.
This macro is invoked in definition 125.

5.4.2 Equivalences

5.4.3 Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block

in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. The scope of the formal parameters is the `ProcedureRange`.

Semantics[145] ≡
```
SYMBOL ProcedureRange INHERITS RangeScope END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

We need to distinguish three roles that may be played in the heading by those identifiers.

The `FormalParamIdTypeDef` is the defining occurrence of the formal parameter's identifier. As discussed earlier, we require that each formal parameter be given a type in order to be able to determine the type of any expression.

A `FormalParameter` is an applied occurrence in the `FormalParameterPart`. It is used to determine the type of argument that will appear in a specific position in the argument list of a call.

Finally, the `ValueParameter` is an applied occurrence in the value list. It allows the analyzer to set the value property of the corresponding formal parameter.

Semantics[146] ≡
```
SYMBOL FormalParamIdTypeDef INHERITS IdentOcc, IdDefScope, MultDefChk END;
SYMBOL FormalParameter      INHERITS IdentOcc, IdUseScope, ChkIdUse   END;
SYMBOL ValueParameter       INHERITS IdentOcc, IdUseScope, ChkIdUse   END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

Formal parameters will, whenever the procedure is activated, be assigned the values of or be replaced by actual parameters. Thus the procedure plays the role of an operator with an operand list. We also need the definition table keys of the formal parameters in order to verify restrictions in Section 4.7.5.

Formal parameter key list[147] ≡
```
Formals: ParameterList; "ParameterList.h"
```

This macro is invoked in definition 16.

Semantics[148] ≡
```
SYMBOL ProcedureDeclaration INHERITS OperatorDefs                END;
SYMBOL FormalParameterPart  INHERITS OpndTypeListRoot, ParameterListRoot END;
SYMBOL FormalParameter      INHERITS OpndTypeListElem, TypedUseId,
                                     ParameterListElem COMPUTE
  SYNT.ParameterElem=THIS.Key;
END;

RULE: ProcedureDeclaration ::= 'procedure' ProcedureIdentifier ProcedureRange
COMPUTE
  ProcedureDeclaration.GotOper=
    ListOperator(
      ProcedureIdentifier.Key,
      NoOprName,
      CONSTITUENT FormalParameterPart.OpndTypeList,
      voidType);
  ProcedureIdentifier.GotProp=
```

```
    ORDER(
      ResetKind(ProcedureIdentifier.Key,ProcedureIdn),
      ResetFormals(
        ProcedureIdentifier.Key,
        CONSTITUENT FormalParameterPart.ParameterList));
END;

RULE: ProcedureDeclaration ::= Type 'procedure' ProcedureIdentifier ProcedureRange
COMPUTE
  ProcedureDeclaration.GotOper=
    ListOperator(
      ProcedureIdentifier.Key,
      NoOprName,
      CONSTITUENT FormalParameterPart.OpndTypeList,
      Type.Type);
  ProcedureIdentifier.GotProp=
    ORDER(
      ResetKind(ProcedureIdentifier.Key,ProcedureIdn),
      ResetFormals(
        ProcedureIdentifier.Key,
        CONSTITUENT FormalParameterPart.ParameterList));
END;
```

This macro is defined in definitions 51, 56, 63, 72, 82, 83, 89, 92, 95, 96, 109, 113, 114, 130, 133, 134, 136, 141, 145, 146, and 148.
This macro is invoked in definition 7.

5.4.4 Values of Function Designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more assignment statements with the procedure identifier in a left part.

This is checked by defining an `InBody` property for each procedure identifier. `InBody` has the value 0 outside of the procedure body and a nonzero value within the body. The `inBody` chain is used to control the value of `InBody`.

Values of Function Designators[149] ≡

```
CHAIN inBody: VOID;
ATTR InBody: int;

SYMBOL ROOTCLASS COMPUTE
  CHAINSTART HEAD.inBody=0;
END;

RULE: ProcedureDeclaration ::= 'procedure' ProcedureIdentifier ProcedureRange
COMPUTE
  ProcedureRange.inBody=
    ResetInBody(ProcedureIdentifier.Key,1) <- ProcedureDeclaration.inBody;
  ProcedureDeclaration.inBody=
    ResetInBody(ProcedureIdentifier.Key,0) <- ProcedureRange.inBody;
END;

RULE: ProcedureDeclaration ::= Type 'procedure' ProcedureIdentifier ProcedureRange
```

```
COMPUTE
  ProcedureRange.inBody=
    ResetInBody(ProcedureIdentifier.Key,1) <- ProcedureDeclaration.inBody;
  ProcedureDeclaration.inBody=
    ORDER(
      IF(NE(GetInBody(ProcedureIdentifier.Key,1),2),
        message(ERROR,"No assignment to the function designator",0,COORDREF)),
      ResetInBody(ProcedureIdentifier.Key,0)) <- ProcedureRange.inBody;
END;

RULE: LeftPart ::= Variable ':=' COMPUTE
  LeftPart.inBody=
    IF(NE(GetInBody(Variable.Key,0),0), ResetInBody(Variable.Key,2))
    <- LeftPart.inBody;
END;
```

This macro is defined in definitions 149 and 151.
This macro is invoked in definition 7.

Properties characterizing function designators[150] ≡

```
InBody: int;
```

This macro is invoked in definition 16.

The type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration.

Values of Function Designators[151] ≡

```
SYMBOL ProcedureDeclaration INHERITS TypedDefinition END;
SYMBOL ProcedureIdentifier  INHERITS TypedDefId      END;

RULE: ProcedureDeclaration ::= 'procedure' ProcedureIdentifier ProcedureRange
COMPUTE
  ProcedureDeclaration.Type=voidType;
END;

RULE: ProcedureDeclaration ::= Type 'procedure' ProcedureIdentifier ProcedureRange
COMPUTE
  ProcedureDeclaration.Type=Type.Type;
END;
```

This macro is defined in definitions 149 and 151.
This macro is invoked in definition 7.

5.4.5 Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, *must* be included. (Note that this is a change from the Revised Report, as discussed in Section 1.)

Specifications[152] ≡

```
SYMBOL FormalSpecification  INHERITS TypedDefinition END;
SYMBOL FormalParamIdTypeDef INHERITS TypedDefId      COMPUTE
```

```
  INH.GotType=ResetKind(THIS.Key,INCLUDING FormalSpecification.kind);
END;

RULE: SpecificationPart LISTOF FormalSpecification END;

RULE: FormalSpecification ::= Specifier FormalIdList ';' COMPUTE
  FormalSpecification.Type=Specifier.Type;
  FormalSpecification.kind=Specifier.kind;
END;

RULE: Specifier ::= 'string' COMPUTE
   Specifier.Type=stringType;
   Specifier.kind=VariableIdn;
END;

RULE: Specifier ::= Type COMPUTE
   Specifier.Type=Type.Type;
   Specifier.kind=VariableIdn;
END;

RULE: Specifier ::= 'array' COMPUTE
   Specifier.Type=realType;
   Specifier.kind=ArrayIdn;
END;

RULE: Specifier ::= Type 'array' COMPUTE
   Specifier.Type=Type.Type;
   Specifier.kind = ArrayIdn;
END;

RULE: Specifier ::= 'label' COMPUTE
   Specifier.Type=labelType;
   Specifier.kind=LabelIdn;
END;

RULE: Specifier ::= 'switch' COMPUTE
   Specifier.Type=labelType;
   Specifier.kind=SwitchIdn;
END;

RULE: Specifier ::= 'procedure' COMPUTE
   Specifier.Type=voidType;
   Specifier.kind=ProcedureIdn;
END;

RULE: Specifier ::= Type 'procedure' COMPUTE
   Specifier.Type=Type.Type;
   Specifier.kind = ProcedureIdn;
END;
```

This macro is defined in definitions 152, 154, and 155.
This macro is invoked in definition 7.

Neither the `FormalParameterPart` nor the `ValuePart` may contain repeated identifiers.

Instantiate additional modules for uniqueness checks[153] ≡
```
$/Prop/Unique.gnrc +instance=Formal :inst
$/Prop/Unique.gnrc +instance=Value  :inst
```

This macro is invoked in definition 19.

Specifications[154] ≡
```
SYMBOL FormalParameter INHERITS FormalUnique COMPUTE
  IF(AND(NOT(THIS.FormalUnique),NE(THIS.Key,NoKey)),
    message(
      ERROR,
      CatStrInd("Repeated formal parameter: ",THIS.Sym),
      0,
      COORDREF));
END;

SYMBOL ValueParameter INHERITS ValueUnique COMPUTE
  IF(AND(NOT(THIS.ValueUnique),NE(THIS.Key,NoKey)),
    message(
      ERROR,
      CatStrInd("Repeated value parameter: ",THIS.Sym),
      0,
      COORDREF));
END;
```

This macro is defined in definitions 152, 154, and 155.
This macro is invoked in definition 7.

There must be a formal parameter for every specification.

Specifications[155] ≡
```
SYMBOL ProcedureHeading COMPUTE
  IF(
      NE(
        CONSTITUENTS FormalParameter.Sym       WITH (int, ADD, ARGTOONE, ZERO),
        CONSTITUENTS FormalParamIdTypeDef.Sym  WITH (int, ADD, ARGTOONE, ZERO)),
    message(ERROR,"Fewer formals than specifications",0,COORDREF));
END;
```

This macro is defined in definitions 152, 154, and 155.
This macro is invoked in definition 7.

5.4.6 Code as Procedure Body

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of the hardware representation, the Revised Report gives no further rules concerning this code language.

We have chosen to define `Code` as a body of text enclosed in braces, possibly with nested text of the same form. A C procedure body or compound statement takes this form, which is recognized by the `Ctext` auxiliary scanner from the Eli library:

Code as Procedure Body [156] ≡

```
Code:   $\{     (Ctext) [mkstr]
```

This macro is invoked in definition 2.

The `{` must be escaped because it is a regular expression operator.

The token processor `mkstr` assigns a unique integer value to each code body regardless of its content. This integer value can be used to access the code body.