A Complete Specification of a Simple Compiler

W. M. Waite

CU-CS-638-93 January 1993

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Complete Specification of a Simple Compiler

W. M. Waite

January 1993

Abstract

This report is a complete description of a simple compiler that implements the Pascal- language described in the book "Brinch-Hansen on Pascal Compilers". It was produced by the Eli compiler construction system from a body of text. The compiler was also generated by Eli from that same body of text. Its purpose is to illustrate a complete Eli specification in the context of an introductory compiler construction text. It follows the structure of Brinch-Hansen's book, commenting on the relationship between the hand-written compiler it describes and the specifications from which an equivalent compiler can be generated by Eli. Using this report, a reader can compare and contrast one approach to building a compiler by hand to an equivalent approach using tools.

Contents

1 What the Pascal- Compiler Does

The Pascal- compiler accepts a program written in a subset of Pascal and produces an equivalent program in the language of an abstract stack-oriented machine ideally suited to the execution of Pascal programs. If the source program is not valid according to the rules of Pascal-, the compiler issues an error report keyed to the coordinates (line and column) at which the problem was detected.

1.1 Translation

Here is a fragment of a program written in Pascal- (declarations of the integer variable m, n, q and r have been omitted):

Pascal- code for Euclid's integer division algorithm[1] ≡

```
{
 { m>=0 and n>0 }
 q:=0; r:=m;
 while r>=n do
   begin r:=r-n; q:=q+1 end;
 { q = quotient of m/n and r = remainder of m/n }
}
```

This macro is NEVER invoked.

The Pascal- compiler will accept a program containing this fragment and produce a program containing an equivalent fragment in the language of the abstract stack machine. In order to make it readable by humans, the abstract machine code is output as a text file. Each line of the file contains a single symbolic operation with arguments enclosed in parentheses (the comments on the right were added to explain the compiler's implementation):

Abstract machine code for Euclid's integer division algorithm[2] ≡

```
{
 Variable(0,5)    Push the address of q onto the stack
 Constant(0)      Push the constant 0 onto the stack
 Assign(1)        Store the value at the address, removing both
 Variable(0,6)    Push the address of r onto the stack
 Variable(0,3)    Push the address of m onto the stack
 Value(1)         Replace the address at the top of the stack with its content
 Assign(1)
 DefAddr(L1)      Represent the current address symbolically by ''L1''
 Variable(0,6)    Push the address of r onto the stack
 Value(1)         Replace the address at the top of the stack with its content
 Variable(0,4)    Push the address of n onto the stack
 Value(1)         Replace the address at the top of the stack with its content
 NotLess          Compare the two values, removing both and setting the test flag
 Do(L2)           If the test flag is false, go to address represented by ''L2''
 Variable(0,6)
 Variable(0,6)
 Value(1)
 Variable(0,4)
 Value(1)
 Subtract         Replace the top two values by (second-top)
 Assign(1)
 Variable(0,5)
```

```
    Variable(0,5)
    Value(1)
    Constant(1)
    Add            Replace the top two values by (second+top)
    Assign(1)
    Goto(L1)       Go to address represented by ``L1''
    DefAddr(L2)    Represent the current address symbolically by ``L2''
    }
```
This macro is NEVER invoked.

This form of the abstract machine code is not only easy for humans to read, thereby allowing them to understand the compiler's translation, but is also easy to implement via C pre-processor macros. Thus the programs produced by the compiler can be run on any machine that provides C.

1.2 Error reporting

If an incorrect program is submitted to the Pascal- compiler, it must deduce that the program *is* incorrect and give the user an indication of the problem. Error reports are keyed to a particular position in the program, specified by a file name, line number, and character position within the line. The report itself consists of a *severity* and a short text describing the problem. All of the errors detected by the Pascal- compiler are `FATAL` – compilation can proceed, but the generated program cannot be run. A `NOTE` does not describe an error, but provides additional information.

Here is an erroneous program that illustrates a variety of reports:

Error reporting example[3] ≡
```
  {
   {Miscellaneous errors}
   program MiscError;
     const
       b = c;
     type
       T = array [5..1] of integer;
       U = record x: true end;
       V = array [false..true] of integer;
     var
       x, y, x: integer;
       z: V;
     begin
     y := 1 and 2;
     y := 2 * (3+4;
     z[1] := &2;
     end.

   "miscerr", line 14:16 FATAL: Syntax error
   "miscerr", line 14:16 NOTE: Parsing resumed here
   "miscerr", line 15:11 FATAL: char '&' (ascii:38) is not a token
   "miscerr", line 4:9 FATAL: identifier not defined
   "miscerr", line 6:16 FATAL: Lower bound may not exceed upper bound
   "miscerr", line 7:19 FATAL: Must be a type identifier
   "miscerr", line 10:5 FATAL: identifier is multiply defined
   "miscerr", line 10:11 FATAL: identifier is multiply defined
   "miscerr", line 13:10 FATAL: Invalid operand for this operator
  "miscerr", line 15:3 FATAL: Invalid index type
  }
```
This macro is NEVER invoked.

All of the reports are for file `miscerr`. At the fourteenth line, the compiler detected a syntax error upon reaching character position 16. Character position 16 contains the semicolon of the assignment statement `y := 2 * (3+4;`. At this point the compiler recognizes that there is no right parenthesis to match the left parenthesis before the `3`. It recovers from the error by assuming that a right parenthesis is present, and continues analysis of the program by accepting the semicolon. (That is the meaning of the `NOTE` following the syntax error report.)

2 A Pascal Subset

The compiler described here accepts a subset of Pascal known as "Pascal-". It has enough features to illustrate the basic techniques for specifying a compiler. With the exception of coercions, the omitted Pascal features do not require any techniques beyond those described here.

This chapter begins by describing the differences between Pascal- and Pascal, assuming that you are already familiar with Pascal. An informal description of the Pascal- vocabulary and a formal description of the structure of a Pascal- program complete the language overview.

2.1 General description

Pascal- has only two simple types, Boolean and integer, and two structured types, array types and record types. Boolean and integer are predefined, with the names `Boolean` and `integer` respectively. Every structured type also has a name, which is introduced by a type definition:

Type definition example[4] ≡
```
{
 type
   table = array [1..100] of integer;
   stack = record contents: table; size: integer end;
}
```
This macro is invoked in definition 21.

A type definition always creates a new type; it can never rename an existing type. The following type definition is therefore incorrect:

Incorrect type definition[5] ≡
```
{
 type number = integer;
}
```
This macro is invoked in definition 21.

All variables must be defined and given a type:

Variable definition example[6] ≡
```
{
 var
   A: table;
   x, y: integer;
}
```

This macro is invoked in definition 21.

The type name must always be used in a variable declaration. In other words, it is not possible to create an "anonymous" type as in the following incorrect variable definition:

Incorrect variable definition[7] ≡
```
{
 var A: array [1..100] of integer;
}
```
This macro is invoked in definition 21.

All constants have simple types. Two predefined identifiers, `true` and `false`, represent the two possible values of type `Boolean`. Values of type `integer` are represented in the usual way. Names for constants can be introduced by constant definitions:

Constant definition example[8] ≡
```
{
 const max = 100; on = true;
}
```
This macro is invoked in definition 21.

Assignment statements, procedure calls, if statements, while statements and compound statements are available in Pascal-. A statement may be empty. There are no labels or goto statements, case statements, repeat statements, for statements or with statements.

Procedure declarations can be nested, and a procedure may have variable and/or value parameters of any type. Procedures cannot be passed to other procedures as parameters, and functions cannot be defined. There are two standard procedures, `read(x)` and `write(x)`. `Read` expects a single variable parameter of type integer, obtains the next integer from the standard input stream, and assigns the value of that integer to the variable parameter. `Write` expects a single value parameter of type integer, and places the value of that integer onto the standard output stream as a complete line.

Here is an algorithm that illustrates most of the features of Pascal-:

Complete example program[9] ≡
```
{
 Program ProgramExample;
 const n=100;
 type table=array[1..n] of integer;
 var A: table; i,x: integer; yes: Boolean;

 procedure search(value: integer; var found: Boolean; var index: integer);
   var limit: integer;
   begin index:=1; limit:=n;
   while index<limit do
     if A[index]=value then limit:=index else index:=index+1;
   found:=A[index]=value
   end;

 begin {input table} i:=1;
 while i<=n do
   begin read(A[i]); i:=i+1 end;
 {test search} read(x);
 while x<>0 do
   begin search(x,yes,i);
```

```
    write(x);
    if yes then write(i);
    read(x);
    end
  end.
  }
```
This macro is invoked in definition 21.

2.2 Vocabulary

The vocabulary of a programming language is made up of *basic symbols* and *comments*. There are three kinds of basic symbols: identifiers, denotations and delimiters. Identifiers are names that are freely chosen by the programmer to represent entities like constants, types, variables and procedures. Denotations represent specific values, according to conventions laid down by the language designer. Delimiters are used to give the program structure.

In Pascal-, an identifier is called a `Name`, and consists of a letter that may be followed by any sequence of letters and digits:

Name examples[10] ≡
```
{
 x LongName S100
 CaseInsensitive CASEINSENSITIVE caseinsensitive
 }
```
This macro is invoked in definition 21.

There is no distinction made between upper and lower case letters in Pascal-, so the three names on the second line of the example are indistinguishable from one another.

A `Numeral` is the only denotation in the vocabulary of Pascal-. It consists of a sequence of decimal digits, and denotes an integer value:

Numeral examples[11] ≡
```
{
 1 123
 }
```
This macro is invoked in definition 21.

There are two kinds of delimiters in Pascal-, *word symbols* and *special symbols*:

Word symbols[12] ≡
```
{
 and array begin const div do else end if mod not
 of or procedure program record then type var while
 }
```
This macro is invoked in definition 21.

Special symbols[13] ≡
```
{
 + - * < = > <= <> >= :=
 ( ) [ ] , . : ; ..
 }
```
This macro is invoked in definition 21.

A word symbol cannot be used as a Name in a Pascal- program.

A comment in Pascal- is an arbitrary sequence of characters enclosed in braces { }. Comments may extend over several lines and may be nested to arbitrary depth:

Comment examples[14] ≡
```
{
 if x>0 then {reserve resource} x:=x-1;
 {This is a {nested} comment
 that extends over two lines}
}
```
This macro is invoked in definition 21.

White space (spaces, tabs and new lines) and comments are called *separators*. Any basic symbol may be preceded by one or more separators, and the program may be followed by zero or more separators.

Word symbols, names and numerals are called *undelimited* basic symbols. Any pair of adjacent undelimited basic symbols must be separated by at least one separator:

Example of basic symbol separation[15] ≡
```
{
 {Incorrect}
   ifx>0thenx:=10divx-1;

 {Correct}
   if x>0
 then{Can divide}x:=10   div x-1;
}
```
This macro is invoked in definition 21.

In this example, separators needed between `if` and `x`, `0` and `then`, `then` and `x`, `0` and `div`, `div` and `x`. The separators can be white space and/or comments, as indicated.

2.3 Grammar

A complete program is called a *sentence* of the programming language in which it is written. The structure of that sentence is described in terms of its component *phrases*. Compilation of a program is based on its phrase structure, as defined by a *grammar* for the programming language. The phrase structure must reflect both the way a program is written and the meaning to be conveyed by that program.

It is convenient to divide a grammar into parts that describe coherent sets of phrases, and then to discuss each part individually. Here is a breakdown appropriate for Pascal-, with two rules describing the program structure given explicitly as examples of the notation:

Grammar[16] ≡
```
{
 Program: 'program' ProgramName ';' BlockBody '.' .

 BlockBody:
   ConstantDefinitionPart
   TypeDefinitionPart
   VariableDefinitionPart
   ProcedureDefinitions
```

```
  CompoundStatement .
```

Constant, type and variable definition grammar[17]
Expression grammar[18]
Statement grammar[19]
Procedure grammar[20]
}

This macro is invoked in definition 36.

The first grammar rule says that a sentence in Pascal- has two component phrases, a `ProgramName` and a `BlockBody`. Delimiters `program`, `;` and `.` are used to separate these phrases in the program text as written.

The `BlockBody` phrase is the description of the algorithm, and consists of definitions and a statement. All of the definitions and the statement together constitute the meaning of the algorithm, while the name of the program does not affect the algorithm in any way. A program is written as the sequence of basic symbols `program`, a name, `;`, a sub-sequence describing the algorithm, and `.`. Thus these two rules reflect both the way the program is written and the underlying meaning.

The Pascal- grammar given here is taken almost verbatim from Brinch-Hansen's description in Section 2.4 of "Brinch-Hansen on Pascal Compilers". His grammar does not make some properties of the program clear, however, and these aspects have been changed. Each such change is discussed in the appropriate section below.

Eli does not permit the extensions to BNF that Brinch-Hansen uses to describe optional and repeated phrases. The rules involving these extensions have been re-written to avoid the extensions. Finally, Eli uses `:` instead of `=` to separate the left and right hand sides of a rule, and uses `/` instead of `|` as the separator for alternative right-hand sides. These notational changes will not be discussed further.

2.3.1 Constant, type and variable definition grammar

Constant, type and variable definitions have similar forms in a Pascal- program: A word symbol describing the kind of definitions to follow, and a list of the definitions themselves. In each case, one or more names are defined. If a particular word symbol is missing, then there are no definitions of that kind.

Notice that the grammar distinguishes between a definition of a name and a use of that name (e.g `ConstantNameDef` and `ConstantNameUse`). This distinction reflects the meaning of the program, rather than the way the program is written. The constant name `max` is always written in the same way, but when it appears to the left of `=` in a `ConstantDefinition` that is a *defining occurrence* and when it appears to the right of `=` in a `ConstantDefinition` that is an *applied occurrence*. All of the properties of a name are deduced from the context of its defining occurrence; those properties are made available at the applied occurrences.

Constant, type and variable definition grammar[17] ≡
{

```
ConstantDefinitionPart: / 'const' ConstantDefinitions .
ConstantDefinitions:
  ConstantDefinition / ConstantDefinitions ConstantDefinition.
ConstantDefinition: ConstantNameDef '=' Constant ';' .
Constant: Numeral / ConstantNameUse .

TypeDefinitionPart: / 'type' TypeDefinitions .
TypeDefinitions:
  TypeDefinition / TypeDefinitions TypeDefinition .
```

```
    TypeDefinition: TypeNameDef '=' NewType ';' .
    NewType: NewArrayType / NewRecordType .

    NewArrayType: 'array' '[' IndexRange ']' 'of' TypeNameUse .
    IndexRange: Constant '..' Constant .

    NewRecordType: 'record' FieldList 'end' .
    FieldList: RecordSection / FieldList ';' RecordSection .
    RecordSection: FieldNameDefList ':' TypeNameUse .
    FieldNameDefList: FieldNameDef / FieldNameDefList ',' FieldNameDef .

    VariableDefinitionPart: / 'var' VariableDefinitions .
    VariableDefinitions:
      VariableDefinition / VariableDefinitions VariableDefinition .
    VariableDefinition: VariableNameDefList ':' TypeNameUse ';' .
    VariableNameDefList:
      VariableNameDef / VariableNameDefList ',' VariableNameDef .
    }
```
This macro is invoked in definition 16.

Brinch-Hansen does not distinguish between defining and applied occurrences of names in his grammar. This distinction is made in the text of Section 6.1 of his book, where scope rules are discussed. When one is specifying a language to a compiler generator like Eli, however, natural language descriptions must be formalized.

2.3.2 Expression grammar

Like most programming languages, Pascal- defines precedence and association rules to eliminate the need for explicit parentheses. For example, the precedence rules ensure that the expression `a+b*c` has the same meaning as `(a+(b*c))`, *not* the same meaning as `((a+b)*c)`. Similarly, the association rules ensure that `a-b-c` has the same meaning as `((a-b)-c)`, *not* the same meaning as `(a-(b-c))`. These rules are embodied in the phrase structure of an expression.

Expression grammar[18] ≡
```
  {
   Expression:
     SimpleExpression /
     SimpleExpression RelationalOperator SimpleExpression .
   RelationalOperator: '<' / '=' / '>' / '<=' / '<>' / '>=' .
   SimpleExpression:
     Term / SignOperator Term / SimpleExpression AddingOperator Term .
   SignOperator: '+' / '-' .
   AddingOperator: '+' / '-' / 'or' .
   Term: Factor / Term MultiplyingOperator Factor .
   MultiplyingOperator: '*' / 'div' / 'mod' / 'and' .
   Factor:
     Numeral /
     VariableAccess /
     '(' Expression ')' /
     NotOperator Factor .

   NotOperator: 'not' .
```

```
    VariableAccess:
      VariableNameUse /
      VariableAccess '[' Expression ']' /
      VariableAccess '.' FieldNameUse .
    }
```
This macro is invoked in definition 16.

Precedence and association rules affect the way the program is written, determining the phrase structure of an expression. Once that phrase structure has been determined, however, precedence and association rules have no further effect on the meaning of the program. As far as the meaning of the program is concerned, it is sufficient to distinguish dyadic expressions, monadic expressions, and expressions without operators. Each operator or operand should therefore be a distinct phrase in the sentence. Brinch-Hansen's grammar does not associate the `not` operator with a phrase, because it uses that operator literally in a rule: `Factor: 'not' Factor`. Here the `not` operator is made a distinct phrase by writing Brinch-Hansen's single rule as two: `Factor: NotOperator Factor .` and `NotOperator: 'not' ..`

The first alternative for `Factor` in Brinch-Hansen's grammar is `Constant`. A `Constant` is defined as being either a `Numeral` or a `ConstantNameUse` (see Section 2.3.1). But notice that a `VariableAccess`, another alternative for `Factor`, could be a `VariableNameUse`. When presented with the `Factor` "x", how can the compiler decide whether this name represents a constant or a variable? The distinction is irrelevant as far as the structure of the program is concerned, and the attempt to make it leads to an ambiguous grammar. Brinch-Hansen discusses this ambiguity in Section 5.5 of his book, and resolves it by altering the grammar to the one given here.

2.3.3 Statement grammar

Statements in Pascal- are separated by semicolons, but the fact that a statement can also be empty means that semicolon can be considered to be a terminator.

Statement grammar[19] ≡
```
  {
   Statement:
     AssignmentStatement /
     ProcedureStatement /
     IfStatement /
     WhileStatement /
     CompoundStatement /
     Empty .

   AssignmentStatement: VariableAccess ':=' Expression .

   ProcedureStatement: ProcedureNameUse ActualParameterList .
   ActualParameterList: / '(' ActualParameters ')' .
   ActualParameters: ActualParameter / ActualParameters ',' ActualParameter .
   ActualParameter: Expression .

   IfStatement:
     'if' Expression 'then' Statement $'else' /
     'if' Expression 'then' Statement 'else' Statement .

   WhileStatement: 'while' Expression 'do' Statement .
```

```
    CompoundStatement: 'begin' Statements 'end' .
    Statements: Statement / Statements ';' Statement .
    }
```
This macro is invoked in definition 16.

Brinch-Hansen's grammar contains the rule `ActualParameter: Expression |VariableAccess ..` This rule is ambiguous, because a `VariableAccess` is one form of an `Expression`. He is trying to distinguish a context in which a value parameter is required from that in which a variable parameter is required. These contexts can only be distinguished on the basis of information drawn from the definitions, and therefore do not affect the structure of the program. In Section 5.5 of his book, Brinch-Hansen alters the grammar to use the rule listed here.

The sequence `$'else'` is used in the definition of an `IfStatement` to avoid another ambiguity: What is the meaning of the phrase `if a then if b then x:=1 else x:=2`? If a is `false`, does `x` remain unaltered or is it set to 2? The rules of Pascal require it to remain unaltered, because an else clause is associated with the closest if. Brinch-Hansen's grammar for Pascal- is ambiguous, and there is no mention of the ambiguity in his book. In the absence of specific guidance, the rules of Pascal are assumed here.

This decision is made explicit in the grammar by the inclusion of the sequence `$'else'`. It specifies that the alternative of which it is a part is only valid if *not* followed by the basic symbol `else`.

2.3.4 Procedure grammar

Like a `Program`, a `ProcedureDefinition` has two constituent phrases. The `ProcedureBlock` defines the algorithm abstracted by the procedure, and the `ProcedureNameDef` defines the name by which that algorithm will be known.

Formal parameters are a part of the abstraction, not a part of the name. Their names are local to the procedure body, whereas the procedure name is defined in the surrounding context.

Procedure grammar[20] ≡
```
  {
   ProcedureDefinitions: / ProcedureDefinitions ProcedureDefinition .
   ProcedureDefinition: 'procedure' ProcedureNameDef ProcedureBlock ';' .

   ProcedureBlock: FormalParameterList ';' BlockBody .

   FormalParameterList: / '(' ParameterDefinitions ')' .
   ParameterDefinitions:
     ParameterDefinition / ParameterDefinitions ';' ParameterDefinition .
   ParameterDefinition:
     'var' ParameterNameDefList ':' TypeNameUse /
     ParameterNameDefList ':' TypeNameUse .
   ParameterNameDefList:
     ParameterNameDef / ParameterNameDefList ',' ParameterNameDef .
  }
```
This macro is invoked in definition 16.

2.4 Examples

Here is a summary of the examples:

Examples[21] ≡
{
Name examples[10]
Numeral examples[11]
Comment examples[14]
Word symbols[12]
Special symbols[13]
Example of basic symbol separation[15]
Constant definition example[8]
Type definition example[4]
Incorrect type definition[5]
Variable definition example[6]
Incorrect variable definition[7]
Complete example program[9]
}
This macro is NEVER invoked.

3 Compiler Organization

Eli embodies a large fraction of the knowledge required to build a compiler, providing an overall design that will satisfy the requirements of almost any translation problem. It assumes a particular decomposition of translation problems, supports the solution of the resulting subproblems, and combines those solutions into a complete program that carries out the desired translation.

The compiler generated by Eli reads a file containing the source program, examining it character-by-character. Character sequences are recognized as basic symbols or discarded, and relationships among the basic symbols are used to build a tree that reflects the structure of the source program. Computations are then carried out over the tree, and the results of these computations output as the target program. Thus Eli assumes that the original translation problem is decomposed into the problems of determining which character sequences are basic symbols and which should be discarded, what tree structure should be imposed on sequences of basic symbols, what computations should be carried out over the resulting tree, and how the computed values should be encoded and output.

There is considerable variability in the specific decompositions for particular translation problems. For example, operator overloading is very simple in Pascal-. Only the relational operators are overloaded, and they can be applied only to Boolean or integer operands. Because of this simplicity, there is no need to treat overload resolution as a separate subproblem of the Pascal- translation problem.

This chapter describes the decomposition of the Pascal- translation problem and gives the general framework within which the subproblems are specified. The complete specifications appear in subsequent chapters.

3.1 Subproblems for Pascal-

Lexical analysis is the process that examines the source program text, recognizing basic symbols and discarding separators. Basic symbols are classified by an integer called a *syntax code*. Each delimiter has a unique syntax code, and there is one syntax code for all names and another for all numerals. A name or numeral is characterized by a single integer value called an *intrinsic attribute* in addition to the syntax code. No intrinsic attribute is computed for a delimiter. As it recognizes each basic symbol, the lexical analyzer passes the information characterizing that basic symbol to the *syntax analyzer*.

Syntax analysis determines the phrase structure of the source program and builds a tree to represent it. Each phrase results in a tree node. If a phrase has component phrases, the nodes representing those component phrases are the children of the node representing the composite phrase. Identifiers and denotations are phrases that have no components, and therefore represent leaves of the tree. Every other phrase corresponds to a rule of the grammar.

Rules like `NotOperator: 'not'.` correspond to phrases with no components, because literals are not considered to be phrases. The phrases corresponding to these rules will therefore result in leaves.

Rules like `AssignmentStatement: VariableAccess ':=' Expression.` correspond to phrases with components. The phrases corresponding to these rules will therefore result in tree nodes that have children. (In the case of `AssignmentStatement: VariableAccess ':=' Expression.` there are two components, and hence two children, because names are considered to be phrases and literals are not.)

The syntax analyzer uses the syntax codes provided by the lexical analyzer to define the basic symbols. When it builds a leaf corresponding to an identifier or a denotation, it stores the value of the intrinsic attribute supplied by the lexical analyzer into that node.

Once the tree is built, the *attribute evaluator* establishes a number of values associated with the tree nodes. For example, each node representing an expression has a value associated with it that specifies the type of result yielded by that expression. Consider a Pascal- expression consisting of an operator and two operands. The type specification for the result in this case is determined by the expression's operator. The type specifications for the two operands are used to verify that the operands are legal for the expression's operator.

Attribute evaluation is the heart of any compiler, where most of the effort and most of the complexity resides. For this reason, specifications of attribute evaluation are usually partitioned by function to make them easier to understand. The major functions of attribute evaluation in Pascal- are scope analysis (determining the relationships between name uses and definitions), type analysis (verifying the context conditions on expressions) and code generation (producing a target program to implement the source code). Each of these functions is treated in a separate chapter.

One of the values established by the attribute evaluator is a tree that represents the target code. The final step in the Pascal- compilation is to linearize and output this tree. That process is carried out by a *program text generator*, invoked on the value produced during attribute evaluation.

Some information is associated with specific contexts in the tree, while other information is associated with specific entities in the program. For example, the type of result yielded by an expression is a property of that expression, and is associated with the node representing the expression in the tree. The type of value yielded by a variable, however, is a property of that variable. Since a particular variable could be used many places in the tree, the information about its type must be available globally. The generated compiler includes a *definition table*, in which arbitrary *properties* can be stored and accessed via *keys*. To make the information about a variable's type available globally, the compiler associates a key with each variable and stores the type information in the definition via that key. The key is therefore a unique internal representation of the variable, and is stored at every use of that variable. Because the key is available at each use, the type information can be accessed at each use.

3.2 How a compiler is specified

Eli focuses the user's attention on the requirements and design decisions that are unique to the Pascal- compiler, having already provided solutions to problems common to all compilers. It accepts descriptions of those requirements and design decisions, and combines them with its understanding of compiler construction problems to produce the Pascal- compiler. Requirements and design decisions can be thought of as specifications of instances of subproblems of the Pascal- translation problem. There are three basic ways in which a user might specify an instance of a subproblem: by analogy ("this problem is the same as problem X"), by description ("this is a problem of class Y, and is characterized as follows...") and by solution("here is a program to solve this problem").

To support specification by analogy, Eli provides a library of solutions to common compiler subproblems. A user must have sufficient understanding of these subproblems to recognize that they have been solved before and to find the solutions in the library. For example, a Pascal- name is defined in a single procedure. If the same name is defined in a nested procedure, the outer definition is "hidden" within the inner procedure. This behavior is common to many programming languages, and a solution for the problem of associating the definition of a name with each use of that name can be solved by analogy by instantiating a module from Eli's library:

Instantiate a module to analyze nested definitions[22] ≡

```
{
$/Tool/lib/Name/Nest.gnrc :inst
}
```

This macro is invoked in definition 23.

To support specification by description, Eli provides a set of notations for characterizing common compiler subproblems. A user must not only be able to recognize the kind of subproblem, but also understand the notation used to characterize problems of that kind. The grammar of Section 2.3 is specified in such a notation – a notation used to describe the phrase structure of a language and thus characterize the syntax analysis subproblem for that language.

To support specification by solution, Eli accepts arbitrary C code that solves a unique compiler subproblem, and incorporates it into the generated translator. A user must not only be able to recognize such subproblems, but also be able to solve them completely. An example of a subproblem of the Pascal-compiler that is specified by solution is that of scanning a comment (Section 4.3.2).

Specifications by analogy and description are two different forms of re-use: A specification by analogy re-uses a particular solution, while a specification by description re-uses a problem-solving method. In each case, re-use simplifies the specification by allowing much to be omitted. Eli provides leverage in direct proportion to the set of compiler subproblems for which it embodies solutions and problem-solving methods. In addition to this kind of leverage, however, Eli supports a paradigm for structuring the specification known as *literate programming.*

The literate programming paradigm suggests that code (or in this case specifications) may need to be organized in one way to support human understanding and in another to support implementation. To solve this dilemma, the user creates a *document* whose organization supports human understanding. Information describing the organization that supports implementation is embedded in this document, which can be processed to yield a printed version for human use or a machine-readable version for implementation.

In its simplest form, the literate programming paradigm allows a user to group together specifications that are related but have different forms. For example, some subproblems of lexical analysis are specified by analogy, others by description, and still others by solution. Literate programming groups all of these specifications together into one file, organizing them so that their tasks and relationships are clear.

The document you are reading is an example of a more sophisticated use of literate programming: Not only are the various specifications that define subproblems of the Pascal- compilation problem organized according to function, but they are interspersed with explanations and commentary.

3.3 The Pascal- compiler

Eli requires that a complete translation problem be described by a single file of type `specs`. A type-`specs` file lists all of the specifications for the subproblems of that translation problem. Subproblems specified by analogy are represented by library references (which might be to the Eli library or to some other library). Subproblems specified either by description or by solution are represented by file references. Here is content of `pascal-.specs`, the file that describes the Pascal- translation problem:

The Pascal- compiler[23] ≡
```
{
        /* Lexical and Syntax Analysis */

Structure.fw
pident.gla :kwd /* Exclude keywords from the finite-state machine */


        /* Scope Analysis */

Scope.fw
```
Instantiate a module to analyze nested definitions[22]
```
$/Tool/lib/Name/NoKeyMsg.gnrc :inst
$/Tool/lib/Name/Unique.gnrc :inst


        /* Type Analysis */

Type.fw
$/Tool/lib/Name/Field.gnrc :inst


        /* A Pascal Computer */

Computer.fw
$/Tool/lib/Tech/LeafPtg.gnrc :inst


        /* Code Generation */

Code.fw
}
```
This macro is NEVER invoked.

Five of the specifications are type-`fw` files, which are documents that follow the literate programming paradigm. `Structure.fw` contains Chapters 1-5; the other documents contain one chapter each. Each of Chapters 4-9 ends with a section that describes the purpose and contents of the specification files generated from that chapter.

Lines that begin with `$/Tool/lib/` extract modules from the Eli library, specifying subproblems by analogy. Each of these library modules is described at the appropriate point in this report.

The line `pident.gla :kwd` is an instruction to Eli to produce a lexical analyzer that behaves in a certain way; it is discussed in Section 4.4.

To create a Pascal- compiler from the set of specifications defined by `pascal-.specs`, one or more of the following requests should be made to Eli:

Requests creating a Pascal- compiler[24] ≡
```
{
pascal-.specs +fold +arg=(input) :stdout
pascal-.specs +fold :exe >pascal.exe
pascal-.specs +fold :source >src
}
```
This macro is NEVER invoked.

The first request asks Eli to create the compiler and run it with the command-line argument `input`. Its purpose is to test the generated compiler against a sample Pascal- program. The second request also asks Eli to create the compiler, but to write the executable version into file `pascal.exe` in the current directory. Its purpose is to create an executable version that can be used independently of Eli itself. The third request writes into directory `src` (which must exist before the request is made) all of the files necessary to create an executable version of the compiler. Its purpose is to create a source code version that can be used independently of both Eli and the computer on which Eli is currently running. Directory `src` will contain C files, header files, a Makefile, and a Configure script that can tailor the other files for specific machine/operating system combinations.

All of the requests specify the parameter `+fold`, which causes Eli to generate a lexical analyzer that is case-insensitive. The meaning of this parameter is further elaborated in Section 4.4.

3.4 Errors and Failures

The compiler must report any errors in the source program. Lexical errors (character sequences that are neither separators nor basic symbols) and syntactic errors (sequences of basic symbols that cannot be built into phrases) are detected automatically by the generated lexical and syntactic analyzers. Contextual errors (such as a Boolean operand for an addition operator) must be detected by tests specified as part of the attribution. When one of these tests detects an error, it must invoke the error reporting module with information about the nature of the error and its location. The location is determined from coordinate information (source program line and character position) stored in the node of the tree at which the computation is being made. Since the coordinate information is always supplied in exactly the same way, it is convenient to define a macro to simplify the error reporting:

report.head[25] $\equiv$
```
{
 #define Message(s,t)    message(s,t,0,COORDREF)
 /* Produce a report at the leftmost symbol of the current fragment
  *    On entry-
  *       s=severity (INFO, WARNING, FATAL, DEADLY)
  *       t=report text
  ***/
}
```
This macro is attached to an output file.

A type-`head` file contains C pre-processor directives that should be supplied to the generated attribution module. All type-`head` files are collected, in some arbitrary order, into a single file called `HEAD.h`. The generated attribution module contains the C pre-processor directive `#include "HEAD.h"`.

4 Lexical Analysis

The purpose of lexical analysis is to determine the sequence of basic symbols making up the source program. Vocabulary errors, such as invalid characters, missing separators and unterminated comments, are detected and reported during lexical analysis. An Eli-generated compiler produces a sequence of basic symbols even if vocabulary errors are found. That sequence contains only basic symbols that are correct according to the language definition, and mirrors the program as closely as possible given the particular errors.

The lexical analyzer scans the input text character-by-character, recognizing basic symbols and discarding separators. It must determine a syntax code for each basic symbol and an intrinsic attribute for each `Name` and `Numeral`.

4.1 Source text

Eli supplies a module for reading the source text and making it available to the remainder of the system. No specification is required for this module. It will be extracted from the Eli library if it is needed, unless the user supplies a module that defines its operations.

The operation `initBuf` initializes the source text module. It has two arguments, the character form of the name of an open file and the descriptor for that file. On return from `initBuf`, `TokenEnd` addresses the first character of the file. If the file is empty, `TokenEnd` addresses a null character (ASCII NUL, with value 0). Otherwise `initBuf` guarantees that the string addressed by `TokenEnd` consists of at least one complete line, including its terminating newline character. If the character following a terminating newline is not the null character then the following line, including its terminating newline character, is also guaranteed to be a part of the string.

When the client of the source module has dealt completely with the line or lines of the input file that are in memory, it invokes the source module operation `refillBuf`. The `refillBuf` operation has one argument, a pointer to the null character terminating the current string. Upon return from `refillBuf`, the conditions described in the previous paragraph hold (i.e. `refillBuf` has the same exit condition as `initBuf`).

The lexical analyzer imposes a coordinate system on the text, with each coordinate consisting of a line number and a column number within the line. All components of the lexical analyzer that move across line boundaries must maintain these coordinates. They are implemented by two variables exported by the source text module but maintained by its clients: `LineNum` and `StartLine`. `LineNum` contains the integer line number, while `StartLine` addresses the character position preceding the first character of the line. Thus the column number can be computed by taking the difference between the pointer to the current character and `StartLine`. Horizontal tab characters, which occupy only one position in the string but possibly many in the source line are handled by decrementing `StartLine` appropriately.

4.2 Intermediate code

The syntax code associated with each basic symbol is relevant only for the interaction between the lexical and syntactic analyzers. It is supplied by Eli, and its value is of no concern. There is no need to specify it.

Each distinct Pascal- `Name` must be given a unique intrinsic attribute value, and instances of the same name must be given the same value. We will want to use this value to associate definitions and uses, so it must be compatible with the module we use for that process. Eli provides a module that can associate definitions with uses, and that module expects each name to be represented by a unique, small integer.

A Pascal- `Numeral` denotes an integer value, and this value will be placed into the target machine code as an integer. It is therefore reasonable to make the intrinsic attribute of a `Numeral` the integer value it denotes.

4.3 Scanning

The delimiters of Pascal- are specified by literals in the grammar. Eli will extract those literals from the grammar, so they need not be re-specified here. Pascal- has two non-literal basic symbols, `Name` and `Numeral`. These must be specified, as must the form of a comment.

Because there are only a few basic symbol formats used in most programming languages, Eli provides a library of *canned descriptions*. These canned descriptions provide not only the format of the basic symbol, but also the processors needed to establish the value of an appropriate intrinsic attribute. They are another example of specifying a subproblem (in this case scanning a name) by analogy.

The Pascal- definition of a name is the same as the Pascal definition of an identifier, so the specification of a `Name` can be stated in terms of a canned description. Numerals and comments are somewhat more complex, and their specifications are discussed below.

Scanning[26] ≡

```
{
Name:           PASCAL_IDENTIFIER
```
Scan a numeral[27]
Scan a comment[29]
```
}
```
This macro is invoked in definition 32.

This specification defines the basic symbol `Name`, which appears to the left of a colon (`:`), by means of the canned description `PASCAL_IDENTIFIER`. `PASCAL_IDENTIFIER` describes a character sequence that begins with a letter and continues with a (possibly empty) sequence of letters and digits. Once this character string has been recognized, the processor `mkidn` will be used to provide it with an intrinsic attribute. (The properties of canned descriptions like `PASCAL_IDENTIFIER` are given in the Eli documentation dealing with lexical analysis.)

4.3.1 Scan a numeral

A Pascal- `Numeral` is defined as a sequence of digits. If the generated scanner simply accepts a sequence of digits as a numeral, however, it will decompose the incorrect phrase `10div 3` into three basic symbols: `10`, `div` and `3`. Although this is probably the programmer's intent, it is a violation of the language definition.

The generated scanner must therefore accept a sequence of digits and then verify that the next character is *not* a letter. If a letter is found, then the scanner must report a "missing separator" error. Finally, a processor must be invoked to compute the value represented by the sequence of digits and deliver that value as an intrinsic attribute. Here is the resulting specification:

Scan a numeral[27] ≡

```
{
Numeral:        $[0-9]+ (CheckSep)        [mkint]
}
```
This macro is invoked in definition 26.

This specification defines the basic symbol `Numeral`, which appears to the left of a colon (`:`), by means of the regular expression `[0-9]+` that accepts a sequence of one or more decimal digits. When the generated scanner encounters a character that is not a decimal digit, it will invoke the auxiliary scanner `CheckSep`, passing a pointer to the first digit of the sequence and an integer giving the length of the sequence. Upon return from `CheckSep`, the processor `mkint` is invoked. It computes the value of the digit sequence and delivers that value as an intrinsic attribute. The processor `mkint` is a library routine, but the auxiliary scanner `CheckSep` must be defined specially for this task.

`CheckSep` is a C routine that uses a table to check the character following the digit sequence. It assumes the ISO standard character set:

Check for a separator[28] ≡

```
{
static char Letter[] = {
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0
  };

  char *
  CheckSep(start, length)
  char * start;    /* start of characters already recognized */
  int length;      /* length of what was already recognized */
  {
    POSITION Now;

    if (Letter[start[length]]){
      Now = curpos; Now.col += length;
      message(FATAL, "Missing separator", 0, &Now);
      }
    return start+length;  /* First unprocessed character */
  }
  }
```
This macro is invoked in definition 33.

Verifying the separator that must follow a `Numeral` is an example of a subproblem specified by solution. It is solved by the C routine `CheckSep`, which obeys the standard interface given in the Eli documentation for auxiliary scanners: accept a pointer to the start of the basic symbol and the number of characters already recognized, and return a pointer to the first character following the basic symbol.

4.3.2 Scan a comment

Comments can be nested in Pascal-, and all comments are delimited by braces. They may span multiple lines. This behavior does not correspond to any canned description, so the comment scanning subproblem must be specified by solution:

Scan a comment[29] ≡
```
  {
                   $"{"    (Comment)
  }
```
This macro is invoked in definition 26.

The auxiliary scanner `Comment` begins with the character following the opening brace that was recognized by the specified regular expression, and seeks the matching closing brace. It invokes itself recursively for a nested comment, handles transitions between lines, and detects an unclosed comment:

Comment scanner[30] ≡
```
  {
   char *
   Comment(start, length)
   char * start;   /* start of characters recognized by reg expr */
   int length;     /* length of what was recognized in reg expr */
   {
     register char c;
     register char *p = start + length; /* first char not yet processed */
```

```
    for (;;) {
      while( (c = *p++) && c != '}' && c != '\n' && c != '{' ) ;
      if(c == '}') return(p);
      else if (c == '\n'){ LineNum++; StartLine = p-1; }
      else if (c == '{') p = Comment(start, p-start);
      else /*if (c == '\0')*/ {   /* end of the current input buffer */
        refillBuf(p-1); p = TokenEnd; StartLine = p-1;
        if (*p == '\0'){
          message(FATAL, "file ends in comment", 0, &curpos);
          return(p);
        }
      }
    }
  }
  }
```
This macro is invoked in definition 33.

`Comment` simply advances `p` until it reaches a "significant" character. If that character signals a new line, the line number is incremented; if it marks the end of the input, additional input is requested from the source module (Section 4.1). The value returned by `Comment` is a pointer to the first character position beyond the end of the comment, as required by the standard auxiliary scanner interface.

4.4 Identifier table

The processor `mkidn` assigns a unique integer for each distinct name. It uses a hash table to check whether the name has been seen before. It stores the string form of each name once in an array of strings, returning the index of the stored string. That index is then made the value of the intrinsic attribute.

Pascal- requires that letter case not be distinguished in identifiers and keywords. Two additional specifications must be provided to obtain this behavior: `mkidn` must be specified case-insensitive and the keywords must be recognized as though they were identifiers. The `+fold` parameter of an Eli request specifies that `mkidn` is to ignore letter case.

Keywords appear as literals in the grammar. Normally, a grammar literal is built into the implementation of the lexical analyzer. To recognize certain literals using the mechanism used to recognize identifiers requires that those literals be removed from the set of literals passed to the lexical analyzer generator. The file `keywords.gla` defines the lexical structure of the Pascal- keywords:

keywords.gla[31] ≡
```
  {
   Name:   PASCAL_IDENTIFIER
  }
```
This macro is NEVER invoked.

By adding the line `keywords.gla :kwd` to the type-`specs` file that defines the Pascal- compiler (Section 3.3), we tell Eli to remove all literals having the structure defined by `keywords.gla` from the set passed to the lexical analyzer generator. It also adds those literals, with their corresponding syntax codes, to the table used by `mkidn`.

4.5 Specification files for lexical analysis

Three kinds of specifications are needed to define the Pascal- lexical analysis problem to Eli.

4.5.1 lexical.gla

A type-`gla` file describes the lexical structure of the comments and the basic symbols that do not appear literally in the grammar. Eli uses this information, in conjunction with the descriptions of the literal basic symbols derived from the grammar, to construct the scanner.

lexical.gla[32] ≡

```
{
 Scanning[26]
}
```

This macro is attached to an output file.

4.5.2 lexical.c

A type-`c` file describes a problem by giving its solution. The problems of checking for a missing separator following a digit string and scanning a Pascal- comment were described in this way.

lexical.c[33] ≡

```
{
 #include "err.h"
 #include "source.h"

 Check for a separator[28]
 Comment scanner[30]
}
```

This macro is attached to an output file.

No interface specification is required for this module, because all auxiliary scanners and processors obey fixed interface conventions. The scanner generator can therefore provide the appropriate `extern` declarations, given the routine names.

5 Syntax Analysis

The purpose of syntax analysis is to determine the phrase structure of the source program, and build a tree with one node per phrase. Structural errors, such as unbalanced parentheses, are detected and reported during syntactic analysis. An Eli-generated compiler constructs a tree even if structural errors are found. That tree's structure is correct according to the language definition, and mirrors the source program as closely as possible given the particular errors.

The source language grammar specifies the phrase structure of any source program, and serves as the basic specification for syntax analysis. It may be necessary to modify the grammar to ensure that it is complete and unambiguous. To make this guarantee, it may be necessary to distinguish some phrases that have identical meanings and should correspond to identical nodes in the tree. These two points, and the specifications resulting from them, are the subject of this chapter.

5.1 Parser construction

Eli generates a particular kind of syntax analyzer, called an *LALR(1) parser*, from the grammar. This is possible only if the grammar satisfies certain conditions: At every point during a left-to-right scan of the source program's basic symbols, the parser must be able to tell whether or not it is at the end of a phrase and, if so, *which* phrase it has just completed.

The Pascal- grammar as defined in Section 2.3 satisfies the LALR(1) condition, but it is not compatible with the definition of the vocabulary given in Section 2.2. This section corrects the incompatibilities.

5.1.1 Distinguishing name contexts

The Pascal- grammar distinguishes several different kinds of names, such as `ConstantNameDef`, `ConstantNameUse` and `VariableNameUse`. Symbols like these are not defined in the grammar, and they are not specified as basic symbols either. A human reader has no difficulty in assuming that they are, in fact, names; this correspondence must be made explicit if the grammar is to specify the parsing process. Each of these symbols (except `ProgramName`) defines a specific context that determines the meaning of the name. These contexts must be represented by distinct nodes in the tree, because of their distinct meanings:

Distinguishing name contexts[34] ≡

```
{
 ProgramName: Name .

 ConstantNameDef: Name .
 ConstantNameUse: Name .

 TypeNameDef: Name .
 TypeNameUse: Name .
 FieldNameDef: Name .
 FieldNameUse: Name .

 VariableNameDef: Name .
 VariableNameUse: Name .

 ProcedureNameDef: Name .
 ProcedureNameUse: Name .
 ParameterNameDef: Name .
}
```

This macro is invoked in definition 36.

5.1.2 Deleting the symbol "Empty"

The symbol `Empty` is used in the grammar to emphasize that an empty statement is allowed, but this symbol is never defined. There is no need to define it, because any meaning attributed to the empty statement would be associated with the tree node representing that statement. Therefore the symbol `Empty` should be replaced in the grammar by nothing at all:

Deleting the symbol "Empty"[35] ≡

```
{
 #define Empty
}
```

This macro is invoked in definition 36.

Grammars are examined by the C pre-processor, as are most Eli specifications. Thus pre-processor directives an C-style comments are allowed in grammars, as they are in virtually every Eli specification.

5.2 Specification files for syntax analysis

A type-`con` file specifies the grammar for the source language. This information is used to construct the parser and define some of the basic symbols.

syntax.con[36] ≡
```
{
  Deleting the symbol "Empty"[35]
  Grammar[16]
  Distinguishing name contexts[34]
  }
```
This macro is attached to an output file.

6 Scope Analysis

A Pascal- program uses names to refer to constants, types, fields, variables, procedures and parameters. Each name must be defined, and then may be used anywhere within the *scope* of that definition. The scope of a definition is a phrase of the program, and within such a phrase there can be only one definition of a particular name. The purpose of scope analysis is to make certain that every name has exactly one definition, and to associate each use of that name with its definition.

It is convenient to split scope analysis into two parts. The first, which associates definitions of names with their uses, is called *consistent renaming.* Once this association has been made, it is easy to report missing definitions and multiple definitions.

6.1 Consistent renaming

The constants, types, fields, variables, procedures and parameters referred to by names are basic components of the algorithm described by the program. Brinch-Hansen calls these basic components *objects.* Each object has properties that are established by that object's definition and examined at each use. It is therefore reasonable to represent each object internally by a unique key that allows access to arbitrary information in a global data base. The consistent renaming process associates the appropriate key with each occurrence of a name, "renaming" it with the key that represents the object referred to by the name.

Definitions and uses of names are distinguished in the grammar. For example, a `ConstantNameDef` phrase is the definition of a constant name and a `TypeNameUse` phrase is a use of a type name. A definition's scope is the smallest *block* containing its definition, excluding the text of nested blocks defining the same name. Three phrases, `ProcedureBlock`, `Program` and `StandardBlock`, are classified as blocks. A `ProcedureBlock` can be nested within another `ProcedureBlock` or within the `Program`. The `StandardBlock` is an imaginary block in which the standard objects are defined.

Scope rules based on nested phrases, in which the scope of a definition is the phrase containing the definition exclusive of nested phrases defining the same name, are common in programming languages. Eli provides a library module to implement consistent renaming according to these scope rules. The consistent renaming problem is therefore solved by analogy. Its solution requires that the user understand the problem, know that an appropriate library module exists and how to use it, and instantiate that library module.

Directory `$/Tool/lib` contains a set of *generic* library modules that solve common subproblems. Each module is a file of type `gnrc`, and is instantiated by requesting a derivation to `:inst`. They are gathered into subdirectories according to the kinds of problems they solve; the consistent renaming problem for nested scopes is solved by the module `$/Tool/lib/Name/Nest.gnrc`. To instantiate this module, therefore, the line `$/Tool/lib/Name/Nest.gnrc :inst` appears in the type-specs file describing the Pascal- compiler (Section 3.3).

`Nest.gnrc` exports four symbols to embody the concepts involved: a `RangeNest` is a phrase that can contain definitions, the `RootNest` is the "outermost" such phrase, an `IdDefNest` is a name definition, and an `IdUseNest` is a name use. Name definitions and uses are assumed to be represented by tree nodes having a `Sym` attribute of type `int` that specifies the corresponding name. (Distinct names result in different values for the `Sym` attribute of the corresponding nodes; identical names result in the same value for the `Sym` attribute of the corresponding nodes.) The module will compute the value of a `Key` attribute of type `DefTableKey` at each tree node representing a name definition or use. `Key` attribute values of associated definitions and uses will be identical. If a use is not associated with any definition, its `Key` attribute value will be the distinguished `DefTableKey` value "`NoKey`".

The library module is used by defining symbols that embody Pascal- scope rule concepts, and attaching the properties of the appropriate library symbols to them by inheritance. Pascal- scope rule concepts include those embodied by the four symbols of the library module, but also include the concept that definitions in a block must be unique and that every name used in a block must have a definition in that block or an enclosing one. These additional concepts do not affect the consistent renaming process, but they mean that Pascal- scope rule concepts are not identical to those of the library module and should therefore be represented by different symbols:

Pascal- scope rule concepts[37] $\equiv$

```
{
 ATTR Key: DefTableKey;

 SYMBOL Block INHERITS RangeNest END;

 ATTR Sym: int;
 SYMBOL NameOccurrence COMPUTE
   SYNT.Sym=CONSTITUENT Name.Sym;
 END;

 SYMBOL NameDef INHERITS IdDefNest, NameOccurrence END;
 SYMBOL NameUse INHERITS IdUseNest, NameOccurrence END;
}
```

This macro is invoked in definition 47.

You will recall that name definition and use phrases were all defined as occurrences of the basic symbol `Name`. During lexical analysis, a unique integer is computed for each name and attached to the tree node representing that basic symbol. This integer must be established as the value of the `Sym` attribute of the tree node representing the name definition or use in order to satisfy an interface condition of the consistent renaming module. `NameOccurrence` embodies this requirement.

The Pascal- concept of the symbol block corresponds directly to the concept represented by the module's `RootNest` symbol. It is true that definitions in the symbol block must be unique, and that every name used in the symbol block must have a definition in that block, but these properties are trivially satisfied because name definition and use in the symbol block is fixed by the language design. Therefore no additional symbol is needed to embody the Pascal- concept of a symbol block; `RootNest` suffices.

6.2 Reporting scope rule violations

The requirements that a definition be unique within its scope and that all uses of names be associated with definitions are common in programming languages, and so Eli provides modules to report violations of them. `$/Tool/lib/Name/Unique.gnrc` reports multiple definitions, and `$/Tool/lib/Name/NoKeyMsg.gnrc` reports uses of undefined names.

It is important to remember that these requirements are distinct from the requirements of consistent renaming, and for that reason are represented by distinct modules. Unique definition is a property of both the phrase in which the definitions occur and the definitions themselves, but it is *not* a property of the uses of a name. Similarly, the requirement that every use of a name be associated with some definition is a property *only* of name uses. Thus the symbols that embody Pascal- scope rules inherit the properties of the symbols exported by the modules as follows:

Reporting scope rule violations[38] ≡

```
{
 SYMBOL Block INHERITS RangeUnique END;
 SYMBOL NameDef INHERITS IdDefUnique END;

 SYMBOL NameUse INHERITS NoKeyMsg END;
}
```

This macro is invoked in definition 47.

6.3 The standard block

Although the programmer regards the standard block as an imaginary block, the compiler must actually implement it in order to make the definitions of the standard objects accessible to the scope analysis algorithms. As far as the scope rules are concerned, the program is nested within the standard block. Therefore the program should be a component phrase of the standard block phrase, and the grammar must be augmented with a production describing this relationship:

The standard block[39] ≡

```
{
 StandardBlock: Program .
}
```

This macro is invoked in definition 46.

6.3.1 Scope rules for the standard block

The standard block is the outermost phrase in which definitions can appear, and this concept is embodied in the symbol `RootNest` exported by the consistent renaming module. `RootNest` is assumed to be represented by a tree node having an attribute `Env`, of type `Environment`, that defines all of the names of standard objects. By default, no standard objects exist. Because standard objects do exist in Pascal-, this default must be overridden by a computation specific to Pascal-:

Scope rules for the standard block[40] ≡

```
{
 SYMBOL StandardBlock: Env: Environment;

 SYMBOL StandardBlock INHERITS RootNest COMPUTE
   SYNT.Env = StandardEnv(NewEnv());
 END;
}
```

This macro is invoked in definition 47.

`NewEnv` is a library routine that creates a value of type `Environment` containing no definitions. `StandardEnv` (described below) populates the environment specified by its argument with definitions of the standard objects of Pascal-.

6.3.2 Standard objects

Pascal- assumes six standard objects: two constants, two types and two procedures. We shall see in later chapters that it is important to be able to refer directly to the keys that represent standard objects. The easiest way to satisfy this requirement is to have a single C module exporting variables containing the keys as well as the operation `StandardEnv` that defines those keys and populates an environment with them. Implementation of the standard objects is thus a problem that we describe by solution.

The exported variable containing the key representing the Pascal- **boolean** type will be named `BooleanKey`, the exported variable containing the key representing the constant **false** will be named `FalseKey`, and so forth. These names will be needed in three different contexts within the module: as external declarations in the module interface, as variable definitions in the module body, and as targets of assignments in the module body. Thus it is convenient to describe the set of standard objects by calls to a C pre-processor macro that can be redefined to suit the context:

Standard objects[41] ≡
```
{
   StdObj(Boolean)
   StdObj(False)
   StdObj(Integer)
   StdObj(Read)
   StdObj(True)
   StdObj(Write)
 }
```
This macro is invoked in definitions 42, 43, and 44.

External declarations in the module interface and variable definitions in the module body have a similar form, but must appear in different files:

External declarations in the module interface[42] ≡
```
{
 #define StdObj(i) extern DefTableKey i/**/Key;
 Standard objects[41]
 #undef StdObj
 }
```
This macro is invoked in definition 49.

Variable definitions in the module body[43] ≡
```
{
 #define StdObj(i) DefTableKey i/**/Key;
 Standard objects[41]
 #undef StdObj
 }
```
This macro is invoked in definition 48.

The variables are assigned their values by the operation that populates the initial environment:

StandardEnv operation[44] ≡
```
{
 #include "termcode.h"

 Environment
 StandardEnv(e)
 Environment e;
```

```
/* Create the standard environment
 *   On entry-
 *     e=empty environment
 *   On exit-
 *     StandardEnv=e populated with the pre-defined identifiers
 ***/
{
  int Code = Name, Attr;

#define StdObj(i) \
  mkidn("i", strlen("i"), &Code, &Attr); i/**/Key = DefineIdn(e, Attr);
```
Standard objects[41]
```
#undef StdObj

  return e;
}
}
```
This macro is invoked in definition 48.

For each standard object, `StandardEnv` obtains the unique integer encoding of the object's name by invoking `mkidn`. This is the routine invoked by the lexical analyzer, via the canned description `PASCAL_IDENTIFIER`, to provide unique encoding for names. Use of the same routine guarantees the same encoding.

The first two arguments to `mkidn` are a pointer to the string to be encoded and the length of that string. The third argument is the syntax code that should be associated with the string on the basis of this instance. If the string has already been coded, this value is replaced by the syntax code previously associated with the string. `Name` is the syntax code associated with Pascal- names by Eli. Its value is given in file `termcode.h`, which Eli generates. Upon return from `mkidn`, the variable defined as the fourth argument has been set to the value of the intrinsic attribute for this string.

`DefineIdn` is a library routine that defines a name in an environment, returning the definition table key associated with that definition. If the given name has not been defined previously in the given environment, a new definition table key is generated and associated with the given name.

6.4 Scope rules for the program

To complete the specification of the Pascal- scope rules, the phrases that are blocks, name definitions and name uses must be provided with the properties associated with those concepts. Since the concepts are represented by symbols, this can be done simply by having the symbols for the phrases inherit from the symbols representing the concepts:

Scope rules for the program[45] $\equiv$
```
{
 SYMBOL Program INHERITS Block END;
 SYMBOL ProcedureBlock INHERITS Block END;

 SYMBOL ConstantNameDef INHERITS NameDef END;
 SYMBOL TypeNameDef INHERITS NameDef END;
 SYMBOL VariableNameDef INHERITS NameDef END;
 SYMBOL ProcedureNameDef INHERITS NameDef END;
 SYMBOL ParameterNameDef INHERITS NameDef END;
SYMBOL ConstantNameUse INHERITS NameUse END;
SYMBOL TypeNameUse INHERITS NameUse END;
SYMBOL VariableNameUse INHERITS NameUse END;
SYMBOL ProcedureNameUse INHERITS NameUse END;
SYMBOL ParameterNameUse INHERITS NameUse END;
}
```

This macro is invoked in definition 47.

Note that the symbols for these phrases will inherit other concepts, and may have additional properties that are unique. These other concepts and properties are defined by other parts of the specification, independent of the properties discussed in this chapter.

6.5 Specification files for scope analysis

Most of the Pascal- scope analysis problem is characterized by analogy, using library modules provided by Eli. The only component of the scope analysis problem characterized by solution is the creation of the standard environment.

Library modules do not require specification files. They are instantiated by requests contained in the list of specification. The relationship between symbols of the library module and symbols of the grammar must, however, be established by specification files.

6.5.1 scope.con

The scope analysis demanded that a physical representation of the fictitious "standard block" be added to the Pascal- grammar. This information is conveyed by a type-`con` file containing the necessary phrase. Eli merges the contents of all type-`con` files to produce the final specification from which the parser is constructed.

scope.con[46] $\equiv$
{
The standard block[39]
}

This macro is attached to an output file.

6.5.2 scope.lido

A type-`lido` file is used to describe the relationships between the library modules and the Pascal- grammar. It also specifies additional computations to take place during attribution. The specifications in type-`lido` files are merged and used to create the attribute evaluator.

scope.lido[47] $\equiv$
{
Pascal- scope rule concepts[37]
Reporting scope rule violations[38]
Scope rules for the standard block[40]
Scope rules for the program[45]
}

This macro is attached to an output file.

6.5.3 scope.c

A type-`c` file implements the solution to a problem that is characterized by solution. Each such file is the code for a single module. Type-`c` files are *not* merged by Eli; each is compiled separately.

scope.c[48] ≡

```
{
 #include "scope.h"

 Variable definitions in the module body[43]
 StandardEnv operation[44]
}
```

This macro is attached to an output file.

6.5.4 scope.h

A type-`h` file defines the interface for a single C module. It uses C pre-processor directives to ensure that it is included in a given program no more than once. Type-`h` files are not merged by Eli.

scope.h[49] ≡

```
{
 #ifndef PREDEF_H
 #define PREDEF_H

 #include "deftbl.h"
 #include "envmod.h"

 External declarations in the module interface[42]

 extern Environment StandardEnv(/* Environment e; */);
 /* Create the standard environment
  *   On entry-
  *     e=empty environment
  *   On exit-
  *     StandardEnv=e populated with the pre-defined identifiers
  ***/
 #endif
}
```

This macro is attached to an output file.

6.5.5 scope.head

A type-`head` file is used to incorporate information into the tree construction and attribution modules. All type-`head` files are merged by Eli into a single file called `HEAD.h`, and this file is included by the tree construction and attribution modules.

scope.head[50] ≡

```
{
 #include "scope.h"
}
```

This macro is attached to an output file.

The operations of the standard environment creation module are made available to the computations described in **source.lido** by including the interface of the standard environment creation module in a type-**head** file (here **scope.head**).

7 Type Analysis

The purpose of type analysis is to determine what kind of object is represented by each name and what type of value each expression yields, and to verify that these satisfy the context conditions of the source language. Errors of agreement, such as use of a type name as an expression operand or use of an integer value to control an if-statement, are detected and reported during type analysis.

Type analysis is an attribution process. Given the **Key** attributes resulting from the consistent renaming, it establishes a **Kind** property for each object and specific additional properties for each type. It also defines a **Type** attribute for each tree node that describes a construct yielding a value.

7.1 Kinds of objects

A Pascal- object belongs to one of seven classes:

Kinds of objects[51] ≡

```
{
  #define Undefined      0
  #define Typex          1
  #define Procedurex     2
  #define Constantx      3
  #define Variable       4
  #define ValueParameter 5
  #define VarParameter   6
}
```

This macro is invoked in definition 88.

This classification is useful because it distinguishes objects that have very different characteristics. Nothing is known about an **Undefined** object, and therefore the compiler should assume whatever characteristics are necessary in the context in which it appears. **Typex** and **Procedurex** objects are allowed only in specific contexts defined by the grammar. The remaining objects are all valid in expression contexts, but the code that must be generated to implement them varies. By grouping them as shown, a simple test can determine whether the context conditions are satisfied.

Brinch-Hansen distinguishes eleven object classes, dividing the **Typex** class into standard types, array types and record types, dividing the **Procedurex** class into standard procedures and user-defined procedures, and adding a class for record fields. The reason he needs this finer classification is that he defines a variant record to store all of the necessary information about each object, and uses the object class to distinguish variants. As we shall see, the information needed to describe an array type is somewhat different from the information needed to describe a standard type or record type. Therefore these two variants must be distinguished, and Brinch-Hansen needs distinct classes to make that distinction.

An Eli-generated compiler stores information about objects in a **definition table**, which is a global database accessed by keys associated with each object. Each item of information is stored as a named *property* of arbitrary type. The property name and type must be declared. Here is a declaration of the **Kind** property, of type **int**, that will be defined for every object:

The object classification property[52] ≡

```
{
Kind: int;
}
```
This macro is invoked in definition 85.

For every property name, Eli creates two definition table operations. One operation (`Get`) is used to query the definition table, the other (`Set`) to update it. Thus the declaration of the `Kind` property causes Eli to create the query operation `GetKind` and the update operation `SetKind`. Both operations take the key of the object of interest as their first argument. The second argument of the query operation `GetKind` is the so-called *default value* for this query: if the object of interest does not have the `Kind` property set, then `GetKind` returns the value of its second argument. `SetKind` has two arguments in addition to the key. The first is the value to which the `Kind` property should be set if the object of interest currently has no `Kind` property set; the second is the value to which the `Kind` property should be changed if the object of interest currently has a `Kind` property set.

Because Pascal- requires definition before use, we can obtain all of the properties of declared objects in a single text-order traversal of the tree. For this purpose, we define a chain `Objects` to represent the invariant "all visible objects have their properties defined". Since the standard names are visible everywhere, this invariant is initially true when all properties of the standard names have been set:

Text-order traversal invariant[53] ≡
```
{
CHAIN Objects: VOID;

SYMBOL StandardBlock COMPUTE
  CHAINSTART HEAD.Objects=StandardIdProperties() DEPENDS_ON THIS.Env;
END;
}
```
This macro is invoked in definition 84.

7.2 Types

Every object that yields a value has a property that specifies the type of the value yielded. Since a type is itself an object, with properties of its own, it is represented by a definition table key:

Types[54] ≡
```
{
Type: DefTableKey;
}
```
This macro is invoked in definition 85.

Each of the standard types `boolean` and `integer` is represented by an object of kind `Typex`. No properties other than the `Kind` are required of standard types by the type analyzer. The `Kind` property is set using the update operation `SetKind` that was created by Eli in response to the property declaration. This operation must be invoked as a part of the implementation of the `StandardIdProperties` operation introduced in the last section:

Set properties of standard names[55] ≡
```
{
   SetKind(IntegerKey,Typex,Undefined);
   SetKind(BooleanKey,Typex,Undefined);
}
```
This macro is defined in definitions 55, 59, and 81.
This macro is invoked in definition 87.

7.2.1 Type declaration

New types can be defined by the user. A type definition constructs either a new array type or a new record type, but in each case provides a definition table key to represent the constructed type:

Type declaration[56] ≡
```
{
 SYMBOL NewType: Key: DefTableKey;

 RULE UserType: TypeDefinition ::= TypeNameDef '=' NewType ';'
 COMPUTE
   NewType.Key=TypeNameDef.Key;
   NewType.Objects=
     SetKind(TypeNameDef.Key,Typex,Undefined)
     DEPENDS_ON TypeDefinition.Objects;
 END;
}
```
This macro is invoked in definition 84.

The specific properties of the constructed type depend on whether it is an array or a record, and will be discussed below. These properties will be stored in the database under the key of the type name, which is made available as the `Key` attribute of `NewType`.

Recall that the `Objects` attribute is a chain representing the assertion "all visible objects have their properties defined". Rule `UserType` therefore asserts that all objects visible at the beginning of the `NewType` phrase have their properties defined if that assertion holds before the type definition, and if the `Kind` property of the type object itself has been set. In this case all of the *known* properties are set, but some of the properties are not known until after the `NewType` phrase has been processed. Thus rule `UserType` cannot guarantee the truth of the assertion to the right of the `TypeDefinition` phrase.

7.2.2 Type name use

The symbol `TypeNameUse` represents a context in which a type name is required. The interesting information carried by that name is its type property, which is delivered to the context as the `Type` attribute of the `TypeNameUse` node.

Type name use[57] ≡
```
{
 SYMBOL TypeNameUse: Type: DefTableKey;

 SYMBOL TypeNameUse COMPUTE
   SYNT.Type=
     IF(EQ(GetKind(THIS.Key,Undefined),Typex),THIS.Key,NoKey)
     DEPENDS_ON THIS.VisibleTypeProperties;
   IF(NE(GetKind(THIS.Key,Typex),Typex),
     Message(FATAL,"Must be a type name"))
     DEPENDS_ON THIS.VisibleTypeProperties;
 END;
}
```
This macro is invoked in definition 84.

`VisibleTypeProperties` asserts that all type names visible at this point have their properties defined. Note that `VisibleTypeProperties` differs from `Objects`, which asserts that *all* visible names (not merely type names) have their properties defined. To see the need for `VisibleTypeProperties`, consider the Pascal- construct `var a, b, c: T;`. The Pascal- compiler computes properties in textual order. But the type represented by `T` is one of the properties of the names `a`, `b` and `c`. Therefore the `TypeNameUse T` must have its `Type` property set *before* all visible names (which include `a`, `b` and `c`!) have their properties defined. Section 7.4.1 shows how the assertion `VisibleTypeProperties` is established for the construct `var a, b, c: T;`.

If the name appearing in this context is not a type name then the `Type` attribute will get the value `NoKey`, used to represent an unknown type. `NoKey` is a distinguished definition table key that can never have any properties. Definition table query operations applied to `NoKey` always yield their default values, and definition table update operations applied to `NoKey` have no effect.

The decisions in the `TypeNameUse` symbol computation illustrate the utility of a distinct default value for each invocation of a query operation. When establishing the value of the `Type` attribute, both an undefined name and an name of the wrong class should yield `NoKey` because in either case the type is unknown. An error, on the other hand, should be reported only if the name is defined but has the wrong class. If the name is not defined, that was a violation of the scope rules and has already been reported. A report that the name must be a type name would be wrong because the compiler does not know that the name in question isn't a type name – it simply has no information about that name's meaning.

7.3 Constants

A constant yields a value, and therefore has a `Type` property in addition to the `Kind` property common to all objects. Because one of the context conditions verified by type analysis is that the lower bound of an array does not exceed the upper bound, and because bounds can be specified by constant names, the actual value of a constant must also be provided as one of its properties:

Constants[58] ≡
```
{
 Value: int;
 }
```
This macro is invoked in definition 85.

There are two pre-defined constant names, `true` and `false`. Their properties must be set as a part of the implementation of the `StandardIdProperties` operation:

Set properties of standard names[59] ≡
```
{
    SetKind(FalseKey,Constantx,Undefined);
    SetType(FalseKey,BooleanKey,NoKey);
    SetValue(FalseKey,0,0);
    SetKind(TrueKey,Constantx,Undefined);
    SetType(TrueKey,BooleanKey,NoKey);
    SetValue(TrueKey,1,0);
 }
```
This macro is defined in definitions 55, 59, and 81.
This macro is invoked in definition 87.

Notice that, although `false` and `true` are Boolean constants, they are given integer values. Integer values are needed because `false` and `true` are legal array bounds. Brinch-Hansen does not discuss the integer values to be provided, but in Figure 7.2 of his book he shows `false` with the value "0". The complete compiler listing he gives in Appendix A.3 sets the values of the Boolean constants to their Pascal ordinal values, which are defined by the Pascal standard to be "0" for `false` and "1" for `true`, so those are the settings used here.

7.3.1 Constant declaration

Constant objects are created by the user via a constant declaration:

Constant declaration[60] ≡
```
{
 RULE ConstDef: ConstantDefinition ::= ConstantNameDef '=' Constant ';'
 COMPUTE
   ConstantDefinition.Objects=
     ORDER(
       SetKind(ConstantNameDef.Key,Constantx,Undefined),
       SetType(ConstantNameDef.Key,Constant.Type,NoKey),
       SetValue(ConstantNameDef.Key,Constant.Value,NoKey))
     DEPENDS_ON ConstantDefinition.Objects;
 END;
 }
```
This macro is invoked in definition 84.

This rule asserts that all visible objects have their properties defined after the constant definition if that assertion holds before the constant definition, and if the `Kind`, `Type` and `Value` properties of the constant definition itself have been set. The use of `ORDER` is a slight overspecification because the property values do not have to be set in any particular order. `ORDER` is being used here simply to group the update operations into a single action asserting that the properties of the constant object have been set.

7.3.2 Constant name use

When a constant is used, the type and value of that constant are determined from the constant's properties and delivered to the surrounding context as attribute values. If the constant is an integer denotation, then its type and value are computed directly; if it is a constant name they are obtained from that name's properties:

Constant name use[61] ≡
```
{
 ATTR Type: DefTableKey;
 ATTR Value: int;

 RULE IntConst: Constant ::= Numeral
 COMPUTE
   Constant.Type=IntegerKey;
   Constant.Value=Numeral.Value;
 END;

 RULE IdnConst: Constant ::= ConstantNameUse
 COMPUTE
   Constant.Type=
     GetType(ConstantNameUse.Key,NoKey) DEPENDS_ON Constant.Objects;
   Constant.Value=
     GetValue(ConstantNameUse.Key,0) DEPENDS_ON Constant.Objects;
   IF(NE(GetKind(ConstantNameUse.Key,Constantx),Constantx),
     Message(FATAL,"Constant name required"))
     DEPENDS_ON Constant.Objects;
 END;
}
```
This macro is invoked in definition 84.

Rule **IdnConst** demonstrates that the assertion that all visible objects have their properties defined is a precondition for the process of accessing the properties of a named constant. Unless that assertion is true, there is no guarantee that the definition table contains valid property values for the object.

Brinch-Hansen's Algorithm 7.1 must verify that the name is a constant name before accessing the property values, in order to ensure that they are defined. If the name is *not* a constant, he then must establish appropriate attribute values with a completely separate set of assignments. Here the fact that the two query operations GetType and GetValue each provide a default value allow us to handle the possibility that the name is not a constant without any additional mechanism.

7.4 Variables

In Pascal- variables and parameters are very similar in their behavior. All yield values, and therefore have a Type property in addition to the Kind property common to all objects. They are defined in groups, with all names in a groups having the same kind and type properties. Finally, a VariableNameUse could actually be a use of a constant name, a variable name or a parameter name.

Variables[62] ≡
```
{
 ATTR Kind: int;
 ATTR Type: DefTableKey;
```
 Variable definitions[63]
 Parameter definitions[64]
 Constant, variable or parameter use[65]
```
}
```
This macro is invoked in definition 84.

7.4.1 Variable definitions

Type is a characteristic of each group of variable names, a fact indicated by attaching the attribute Type to VariableDefinition. A remote attribute access (INCLUDING) is used to make it available at each individual variable name definition. Notice that the remote attribute access abstracts from the details of the tree structure, requiring only that a VariableNameDef node be a descendent of a VariableDefinition node.

Variable definitions[63] ≡
```
{
 RULE VarType: VariableDefinition ::= VariableNameDefList ':' TypeNameUse ';'
 COMPUTE
   VariableDefinition.Type=TypeNameUse.Type;
   TypeNameUse.VisibleTypeProperties=VariableDefinition.Objects;
 END;

 SYMBOL VariableNameDef COMPUTE
   THIS.Objects=
     ORDER(
       SetKind(THIS.Key,Variable,Undefined),
       SetType(THIS.Key,INCLUDING VariableDefinition.Type,NoKey))
```

```
        DEPENDS_ON THIS.Objects;
    END;
    }
```
This macro is invoked in definition 62.

Recall from Section 7.2 that the `VisibleTypeProperties` attribute of `TypeNameUse` asserts that all type names visible at this point have their properties defined. That assertion is true if *all* names visible at the beginning of the `VariableDefinition` phrase have their properties defined (`VariableDefinition.Objects`), because no new type names become visible within the `VariableNameDefList` phrase.

7.4.2 Parameter definitions

Parameters are declared in much the same way as variables, except that the classification of the parameter (as well as its type) is characteristic of the group rather than being fixed. Two attributes, `Kind` and `Type`, are therefore attached to `ParameterDefinition` and accessed remotely by the computations associated with `ParameterNameDef` nodes.

Parameter definitions[64] ≡
```
   {
    RULE ValParmDef:
      ParameterDefinition ::= ParameterNameDefList ':' TypeNameUse
    COMPUTE
      ParameterDefinition.Kind=ValueParameter;
      ParameterDefinition.Type=TypeNameUse.Type;
      TypeNameUse.VisibleTypeProperties=ParameterDefinition.Objects;
    END;

    RULE VarParmDef:
      ParameterDefinition ::= 'var' ParameterNameDefList ':' TypeNameUse
    COMPUTE
      ParameterDefinition.Kind=VarParameter;
      ParameterDefinition.Type=TypeNameUse.Type;
      TypeNameUse.VisibleTypeProperties=ParameterDefinition.Objects;
    END;

    SYMBOL ParameterNameDef COMPUTE
      THIS.Objects=
        ORDER(
          SetKind(THIS.Key,INCLUDING ParameterDefinition.Kind,Undefined),
          SetType(THIS.Key,INCLUDING ParameterDefinition.Type,NoKey))
        DEPENDS_ON THIS.Objects;
    END;
    }
```
This macro is invoked in definition 62.

7.4.3 Constant, variable or parameter use

Variable and parameter names are used in the same context: the name component of a `VariableAccess` phrase. (Constant names can also appear in this context, as well as in other contexts.) The integer values used in the object classification scheme are chosen so that a single test suffices to differentiate constants, variables and parameters from types and procedures.

Because a `VariableAccess` phrase describes a construct that yields a value, it has a `Type` attribute. It also has a `Kind` attribute to permit the compiler to verify that (for example) a constant does not appear on the left-hand side of an assignment.

Constant, variable or parameter use[65] ≡

```
{
 RULE VarIdn: VariableAccess ::= VariableNameUse
 COMPUTE
   VariableAccess.Kind=
     GetKind(VariableNameUse.Key,Variable) DEPENDS_ON VariableAccess.Objects;
   VariableAccess.Type=
     GetType(VariableNameUse.Key,NoKey) DEPENDS_ON VariableAccess.Objects;
   IF(LT(VariableAccess.Kind,Constantx),
     Message(FATAL,"Constant, variable or parameter name required"));
 END;
}
```

This macro is invoked in definition 62.

Note that the computations of the `Kind` and `Type` attribute of the `VariableAccess` both depend on the assertion that all names visible at this point have their properties set. The need for this dependence is clear: Both attributes are properties of the name being used, and might be meaningless if the assertion were false.

Choice of `Variable` as the default value of the `Kind` attribute avoids a spurious error report if the name is undefined. Remember that scope analysis has already reported an error if the name is undefined. If `Undefined` were chosen as the default value, the error "Constant, variable or parameter name required"would be reported here. That report is spurious, because the compiler does not *know* that the name in question is not a constant, variable or parameter name – it has no information about the name.

7.5 Arrays

An array type has four properties in addition to the normal properties of a type object:

Arrays[66] ≡

```
{
 IndexType, ElementType: DefTableKey;
 LowerBound, UpperBound: int;
}
```

This macro is invoked in definition 85.

All of these properties are used in checking the context conditions for expressions.

7.5.1 Array type definition

Array types are declared by specifying the index range and the element type. Each new array type is distinct from all others, even though two array type declarations may look the same.

Array type definition[67] ≡

```
{
 RULE ArrayDef:
   NewArrayType ::= 'array' '[' IndexRange ']' 'of' TypeNameUse
 COMPUTE
```

```
  NewArrayType.Objects=
    ORDER(
      SetLowerBound(INCLUDING NewType.Key,IndexRange.LowerBound,0),
      SetUpperBound(INCLUDING NewType.Key,IndexRange.UpperBound,0),
      SetIndexType(INCLUDING NewType.Key,IndexRange.IndexType,NoKey),
      SetElementType(INCLUDING NewType.Key,TypeNameUse.Type,NoKey))
    DEPENDS_ON NewArrayType.Objects;
  TypeNameUse.VisibleTypeProperties=NewArrayType.Objects;
END;

SYMBOL IndexRange:
  LowerBound, UpperBound: int,
  IndexType: DefTableKey;

RULE Bounds: IndexRange ::= Constant '..' Constant
COMPUTE
  IndexRange.LowerBound=Constant[1].Value;
  IndexRange.UpperBound=Constant[2].Value;
  IndexRange.IndexType=
    IF(EQ(Constant[1].Type,Constant[2].Type),
      Constant[1].Type,
      ORDER(
        IF(AND(NE(Constant[1].Type,NoKey),NE(Constant[2].Type,NoKey)),
          Message(FATAL,"Bounds must be of the same type")),
        NoKey));
  IF(GT(Constant[1].Value,Constant[2].Value),
    Message(FATAL,"Lower bound may not exceed upper bound"));
END;
}
```

This macro is invoked in definition 84.

The assertion that "all visible names have their properties set" is true after the array type declaration if it is true before and if the array type name has its properties set.

7.5.2 Array element use

An array variable can be accessed as a whole simply by stating the name of the variable. One element of an array can also be accessed as a variable by specifying a subscript expression that selects the desired element.

Array element use[68] ≡

```
{
RULE Index: VariableAccess ::= VariableAccess '[' Expression ']'
COMPUTE
  VariableAccess[1].Kind=VariableAccess[2].Kind;
  VariableAccess[1].Type=
    GetElementType(VariableAccess[2].Type,NoKey)
    DEPENDS_ON VariableAccess[1].Objects;
  IF(EQ(VariableAccess[1].Type,NoKey),
    Message(FATAL,"Indexed variable must be of array type"));
  Expression.ExpectedType=GetIndexType(VariableAccess[2].Type,NoKey)
    DEPENDS_ON VariableAccess[1].Objects;
  END;
}
```
This macro is invoked in definition 84.

A constant cannot be of an array type because there is no way to describe an array denotation in Pascal-. Therefore a single test of the type yielded by the indexed object suffices to determine the legality of an indexed selector.

7.6 Records

A record variable consists of some number of distinct *field variables*. Field variable names do not follow the same scope rules as other names. If a field variable name is defined within a record of type `T`, then the scope of that definition is the set of names following the "." in all phrases `VariableAccess '.' FieldNameUse` for which `VariableAccess` yields an object of type `T`.

Scope rules of this kind are common in programming languages, and they cannot be verified during scope analysis because they depend on type analysis. Eli provides a generic library module `$/Tool/lib/Name/Field.gnrc` to implement consistent renaming according to this rule.

`Field.gnrc` exports four symbols to embody the concepts involved: a `FieldScope` is a phrase that contains definitions, a `FieldDef` is a name definition, and a `FieldUse` is a name use. `RootField` is a phrase containing all of the `FieldScope` phrases in the source program. Name definitions and uses are assumed to be represented by tree nodes having a `Sym` attribute of type `int` that specifies the corresponding name. The module will compute the value of a `Key` attribute of type `DefTableKey` at each tree node representing a name definition or use. `Key` attribute values of associated definitions and uses will be identical. If a use is not associated with any definition, its `Key` attribute will be the distinguished `DefTableKey` value "`NoKey`".

7.6.1 Record type definition

Each `FieldScope` must be provided with a `Key` attribute of type `DefTableKey` by some mechanism outside of the module. The same `DefTableKey` value must be provided as the `ScopeKey` attribute of each `FieldUse`, again via a mechanism outside of the module. It is this `DefTableKey` value that links the field with the record in which it is defined.

In Pascal-, the `DefTableKey` value linking records and their field variables is the record's type. The `FieldScope` concept is embodied in the `NewRecordType` phrase, `FieldDef` is embodied in `FieldNameDef` and `FieldUse` is embodied in `FieldNameUse`. Since the field names must be unique within a record and every field variable must be defined, these symbols also inherit from the error reporting modules discussed in Section 6.2.

Record type definition[69] ≡
```
{
 SYMBOL Program INHERITS RootField END;
 SYMBOL NewRecordType INHERITS FieldScope, RangeUnique COMPUTE
   INH.Key=INCLUDING NewType.Key;
 END;

 RULE RecSect: RecordSection ::= FieldNameDefList ':' TypeNameUse
 COMPUTE
   RecordSection.Type=TypeNameUse.Type;
   TypeNameUse.VisibleTypeProperties=RecordSection.Objects;
 END;
ATTR ScopeKey: DefTableKey;

SYMBOL FieldNameDef INHERITS FieldDef, NameOccurrence, IdDefUnique COMPUTE
  INH.ScopeKey=INCLUDING NewType.Key;
  SYNT.Objects=
    SetType(THIS.Key,INCLUDING RecordSection.Type,NoKey)
    DEPENDS_ON THIS.Objects;
END;
}
```

This macro is invoked in definition 84.

The assertion that "all visible names have their properties set" is true after the field declaration if it is true before and if the field name has its properties set.

7.6.2 Field name use

A record variable can be accessed as a whole simply by stating the name of the variable. One field of a record can also be accessed as a variable by specifying a name that selects the desired field.

Field name use[70] ≡

```
{
SYMBOL FieldNameUse INHERITS FieldUse, NameOccurrence, NoKeyMsg END;

RULE Select: VariableAccess ::= VariableAccess '.' FieldNameUse
COMPUTE
  VariableAccess[1].Kind=VariableAccess[2].Kind;
  VariableAccess[1].Type=GetType(FieldNameUse.Key,NoKey)
    DEPENDS_ON VariableAccess[1].Objects;
  FieldNameUse.ScopeKey=VariableAccess[2].Type;
END;
}
```

This macro is invoked in definition 84.

`VariableAccess[1].Type` is set to a property of the field name being used, and might be meaningless unless all visible identifiers have their properties set. Hence its dependence on the assertion `VariableAccess[1].Objects`.

A constant cannot be of a record type because there is no way to describe an record denotation in Pascal-. Therefore the test for an undefined field variable suffices to determine the legality of a field selector.

7.7 Expressions

Type analysis of a Pascal- expression determines the type yielded by every expression and the type required by the context in which that expression appears. It then verifies that the yielded type is appropriate for the context:

Expressions[71] ≡

```
{
ATTR Type, ExpectedType: DefTableKey;
```

```
SYMBOL Expression COMPUTE
  IF(AND(
    AND(NE(THIS.Type,THIS.ExpectedType),NE(THIS.Type,NoKey)),
    NE(NoKey,THIS.ExpectedType)),
    Message(FATAL,"Type yielded is not compatible with the context"));
END;
```

Expression contexts[73]
Operator definitions[76]
}

This macro is invoked in definition 84.

Note that no error report is issued when nothing is known about the expression type (i.e. the type is `NoKey`).

This computation should be applied at *all* nodes representing expressions. In the grammar, however, phrases that describe expressions were given different names in order to make operator precedence and association rules explicit. Operator precedence and association rules only affect the structure of the tree, however, not the meaning of the computation. Thus `A+B` and `A*B` are both dyadic expressions, in which two operands are combined by an operator according to certain rules to yield a single result, even though the first constitutes a `SimpleExpression` and the second a `Term`.

To avoid spurious distinctions among nodes, define *equivalence classes* of phrases:

Equivalence classes of expression and operator phrases[72] ≡
{

```
Expression ::= SimpleExpression Term Factor.
Binop ::= RelationalOperator AddingOperator MultiplyingOperator.
Unop ::= SignOperator NotOperator.
```

}

This macro is invoked in definition 86.

Each equivalence class is represented by the name of one of its member phrases, the one appearing to the left of "`::=`". The name used to represent the members of the equivalence class need not actually appear in the grammar (`Expression` is a phrase in the Pascal- grammar, but `Binop` and `Unop` are not). Every tree node corresponding to a member of an equivalence class must be referred to by the name used to represent that class.

7.7.1 Expression contexts

The `Type` and `ExpectedType` attributes of every expression must be set in order to verify the context condition. `Type` depends on the content of the expression itself, while `ExpectedType` depends upon how the expression is used. In Pascal-, there are four distinct ways of forming an expression:

Expression contexts[73] ≡
{

```
RULE VariableExpr: Expression ::= VariableAccess
COMPUTE
  Expression.Type=VariableAccess.Type;
END;

RULE Denotation: Expression ::= Numeral
COMPUTE
  Expression.Type=IntegerKey;
```

```
END;

ATTR Needs: DefTableKey;

RULE Monadic: Expression ::= Unop Expression
COMPUTE
  Expression[1].Type=Unop.Type;
  Expression[2].ExpectedType=Unop.Needs;
END;

ATTR Left: DefTableKey;

RULE Dyadic: Expression ::= Expression Binop Expression
COMPUTE
  Expression[1].Type=Binop.Type;
  Expression[2].ExpectedType=Binop.Needs;
  Binop.Left=Expression[2].Type;
  Expression[3].ExpectedType=Binop.Needs;
END;
}
```

This macro is invoked in definition 71.

When an expression is formed using an operator, the type yielded by the resulting expression is completely determined by the operator. The `Type` attribute of the operator specifies that type. The `Needs` attribute of the operator specifies the type that must be yielded by the operand(s) (both operands of a dyadic operator must yield the same type in Pascal-). `Needs` is completely determined by the operator in most cases, but depends on `Left` for relational operators.

7.7.2 Operator type information

Every Pascal- operator is described by a phrase in the grammar, and therefore corresponds to a node of the tree. The computations associated with those nodes simply establish the values of the `Needs` and `Type` attributes of the operator. Each phrase needs a rule with a name, and the rule must give the name of the phrase (`Binop` or `Unop`), and the operator indication (`'+'`, `'div'`, etc.) Here is a template that creates the necessary boilerplate:

Operator type information[74](◇5) ≡

```
{
 RULE r◇1: ◇2 ::= ◇3
 COMPUTE
   ◇2.Needs=◇4Key; ◇2.Type=◇5Key;
 END;
 }
```

This macro is invoked in definitions 76, 76, 76, 76, 76, 76, 76, 76, 76, and 76.

Relational operators do not fit this model, because they are *overloaded*: Their operands might be either integer values or Boolean values. Therefore the `Needs` attribute of a relational operator depends upon the types of the operands actually provided to it, while the `Needs` attributes of other operators are determined by the operators.

Relational operator type information[75](◇2) ≡

```
{
 RULE r◇1: Binop ::= ◇2
COMPUTE
  Binop.Needs=IF(EQ(Binop.Left,BooleanKey),BooleanKey,IntegerKey);
  Binop.Type=BooleanKey;
END;
}
```
This macro is invoked in definitions 76, 76, 76, 76, 76, and 76.

Given these templates, the characteristics of the operators are simply listed:

Operator definitions[76] ≡
```
{
 Relational operator type information[75](‘Lss’,‘'<'’)
 Relational operator type information[75](‘Leq’,‘'<='’)
 Relational operator type information[75](‘Gtr’,‘'>'’)
 Relational operator type information[75](‘Geq’,‘'>='’)
 Relational operator type information[75](‘Equ’,‘'='’)
 Relational operator type information[75](‘Neq’,‘'<>'’)

 Operator type information[74](‘Add’,‘Binop’,‘'+'’,‘Integer’,‘Integer’)
 Operator type information[74](‘Sub’,‘Binop’,‘'-'’,‘Integer’,‘Integer’)
 Operator type information[74](‘Mul’,‘Binop’,‘'*'’,‘Integer’,‘Integer’)
 Operator type information[74](‘Div’,‘Binop’,‘'div'’,‘Integer’,‘Integer’)
 Operator type information[74](‘Mod’,‘Binop’,‘'mod'’,‘Integer’,‘Integer’)

 Operator type information[74](‘Neg’,‘Unop’,‘'-'’,‘Integer’,‘Integer’)
 Operator type information[74](‘Nop’,‘Unop’,‘'+'’,‘Integer’,‘Integer’)

 Operator type information[74](‘And’,‘Binop’,‘'and'’,‘Boolean’,‘Boolean’)
 Operator type information[74](‘Or’,‘Binop’,‘'or'’,‘Boolean’,‘Boolean’)
 Operator type information[74](‘Not’,‘Unop’,‘'not'’,‘Boolean’,‘Boolean’)
}
```
This macro is invoked in definition 71.

7.8 Statements

As far as type analysis is concerned, the only effect of a statement is to establish a context for the expressions it contains:

Statements[77] ≡
```
{
 RULE Assign: Statement ::= VariableAccess ':=' Expression
 COMPUTE
   Expression.ExpectedType=VariableAccess.Type;
 END;

 RULE While: Statement ::= 'while' Expression 'do' Statement
 COMPUTE
   Expression.ExpectedType=BooleanKey;
 END;

 RULE OneSided: Statement ::= 'if' Expression 'then' Statement
 COMPUTE
```

```
  Expression.ExpectedType=BooleanKey;
END;

RULE TwoSided: Statement ::= 'if' Expression 'then' Statement 'else' Statement
COMPUTE
  Expression.ExpectedType=BooleanKey;
END;
}
```

This macro is invoked in definition 84.

Procedure statements are deferred to the next section.

Because there is no need to distinguish the different kinds of statement described in the grammar, they are all placed into an equivalence class:

Equivalence classes of statement phrases[78] ≡

```
{
Statement ::= AssignmentStatement ProcedureStatement IfStatement WhileStatement.
}
```

This macro is invoked in definition 86.

7.9 Procedures

Type analysis of procedures verifies that the the arguments of a procedure call agree with the parameter specifications of the procedure called. Since the specification of a parameter is available via its definition table key, a procedure must have a property that specifies its parameters, in addition to the `Kind` property common to all objects:

Procedures[79] ≡

```
{
Formals: KeyArray;
}
```

This macro is invoked in definition 85.

A `KeyArray` is an abstract data type provided by the Eli library to represent fixed-sized, ordered collections of objects. `KeyArray` exports four operations: `NewKeyArray(n)` returns a new collection of size `n` whose elements are indexed by the integers 0 through n-1 inclusive. Initially, all elements of the collection represented by `NewKeyArray(n)` are the distinguished definition table key `NoKey`. Element `i` of collection `a` is set to the value `k` by the operation `StoreKeyInArray(a,i,k)` and its value is obtained by the operation `FetchKeyFromArray(a,i)`. `KeyArraySize(a)` yields the number of elements in a collection, and `NoKeyArray` is the empty collection.

`KeyArray` is an abstract data type, not a generic module. Therefore it need not be instantiated, but its interface must be made available:

Definition table key array module interface[80] ≡

```
{
#include "keyarray.h"
}
```

This macro is invoked in definition 89.

The properties of the standard procedures are established as follows:

Set properties of standard names[81] ≡

```
{
  { KeyArray Formals; DefTableKey FormalKey;

    SetKind(ReadKey,Procedurex,NoKey);
    Formals = NewKeyArray(1);
    FormalKey = NewKey();
    SetKind(FormalKey,VarParameter,Undefined);
    SetType(FormalKey,IntegerKey,NoKey);
    StoreKeyInArray(Formals,0,FormalKey);
    SetFormals(ReadKey,Formals,NoKey);

    SetKind(WriteKey,Procedurex,NoKey);
    Formals = NewKeyArray(1);
    FormalKey = NewKey();
    SetKind(FormalKey,ValueParameter,Undefined);
    SetType(FormalKey,IntegerKey,NoKey);
    StoreKeyInArray(Formals,0,FormalKey);
    SetFormals(WriteKey,Formals,NoKey);
  }
}
```
This macro is defined in definitions 55, 59, and 81.
This macro is invoked in definition 87.

7.9.1 Procedure definitions

The formal parameters of a procedure are characteristic of the `ProcedureBlock` component of the procedure definition, so `ProcedureBlock` is given a `Formals` attribute. Its value is a key array large enough to hold the collection of parameters. A computation at each parameter stores the parameter's definition table key into the key array, and the presence of the key array is asserted when all parameter keys have been stored.

Procedure definitions[82] ≡
```
{
 ATTR Formals: KeyArray;

 RULE ProcDef:
   ProcedureDefinition ::= 'procedure' ProcedureNameDef ProcedureBlock ';'
 COMPUTE
   ProcedureBlock.Objects=
     ORDER(
       SetKind(ProcedureNameDef.Key,Procedurex,Undefined),
       SetFormals(ProcedureNameDef.Key,ProcedureBlock.Formals,NoKeyArray))
     DEPENDS_ON ProcedureNameDef.Objects;
 END;

 CHAIN Counter: int;

 RULE ProcBlock: ProcedureBlock ::= FormalParameterList ';' BlockBody
 COMPUTE
   CHAINSTART FormalParameterList.Counter=0;
   FormalParameterList.Formals=NewKeyArray(FormalParameterList.Counter);
   ProcedureBlock.Formals=
    FormalParameterList.Formals
    DEPENDS_ON FormalParameterList CONSTITUENTS ParameterNameDef.GotFormal;
END;

SYMBOL ParameterNameDef COMPUTE
  SYNT.GotFormal=
    StoreKeyInArray(
      INCLUDING FormalParameterList.Formals,
      THIS.Counter,
      THIS.Key);
  THIS.Counter=ADD(THIS.Counter,1);
END;
}
```
This macro is invoked in definition 84.

7.9.2 Procedure calls

The corresponding formal parameter list is characteristic of the argument list in a call, so `ActualParameterList` is given a `Formals` attribute. The key of the parameter corresponding to each argument is extracted from the key array by the computation at the `ActualParameter` node. The argument type is the type required by the argument expression's context, and thus becomes the value of that expression's `ExpectedType` attribute. If the parameter is a `var` parameter, however, the corresponding argument must also be a variable. Two more attributes, `ExpectedVar` and `IsVariable`, associated with `Expression` nodes verify this condition:

Procedure calls[83] ≡
```
{
 RULE Call: Statement ::= ProcedureNameUse ActualParameterList
 COMPUTE
   CHAINSTART ActualParameterList.Counter=0;
   ActualParameterList.Formals=
     GetFormals(ProcedureNameUse.Key,NoKeyArray)
     DEPENDS_ON Statement.Objects;
   IF(NE(GetKind(ProcedureNameUse.Key,Undefined),Procedurex),
     Message(FATAL,"Procedure name required here"),
     IF(NE(
         ActualParameterList.Counter,
         KeyArraySize(ActualParameterList.Formals)),
       Message(FATAL,"Number of arguments differs from number of parameters")))
     DEPENDS_ON Statement.Objects;
 END;

 RULE Argument: ActualParameter ::= Expression
 COMPUTE
   ActualParameter.Key=
     FetchKeyFromArray(
       INCLUDING ActualParameterList.Formals,
       ActualParameter.Counter);
   ActualParameter.Counter=ADD(ActualParameter.Counter,1);
   Expression.ExpectedType=GetType(ActualParameter.Key,NoKey);
   Expression.ExpectedVar=EQ(GetKind(ActualParameter.Key,NoKey),VarParameter);
 END;
SYMBOL Expression: ExpectedVar, IsVariable: int;

SYMBOL Expression COMPUTE
  INH.ExpectedVar=0;
  SYNT.IsVariable=0;
  IF(AND(THIS.ExpectedVar,NOT(THIS.IsVariable)),
    Message(FATAL,"A variable is required here"));
END;

RULE VariableExpr: Expression ::= VariableAccess
COMPUTE
  Expression.IsVariable=NE(VariableAccess.Kind,Constantx);
END;
}
```
This macro is invoked in definition 84.

`ExpectedVar` and `IsVariable` are both false for almost all expressions. The only case in which `ExpectedVar` is true is when the `Expression` is an argument corresponding to a `var` parameter, and the only case in which `IsVariable` is true is when the `Expression` is a `VariableAccess` that is not a constant. Computations in these two contexts override the normal computations associated with each `Expression` node.

7.10 Specification files for type analysis

Most of the type analysis problem is characterized by attribute computations. These computations involve both attributes and properties, which must be defined. Properties of standard names are set by a C module.

7.10.1 type.lido

Attribute computations are specified in a type-`lido` file. Eli merges this file with all other type-`lido` files and uses the resulting specification to create the attribute evaluator.

type.lido[84] ≡

{
Text-order traversal invariant[53]
Constant declaration[60]
Constant name use[61]
Type declaration[56]
Type name use[57]
Array type definition[67]
Array element use[68]
Record type definition[69]
Field name use[70]
Variables[62]
Expressions[71]
Statements[77]
Procedure definitions[82]
Procedure calls[83]
}

This macro is attached to an output file.

7.10.2 type.pdl

Properties are declared in a type-`pdl` file. Eli merges this file with all other type-`pdl` files and uses the resulting specification to create the definition table module.

type.pdl[85] ≡

```
{
 "keyarray.h"
 The object classification property[52]
 Constants[58]
 Types[54]
 Arrays[66]
 Procedures[79]
 }
```

This macro is attached to an output file.

7.10.3 pascal-.sym

A type-`sym` file specifies equivalence classes of phrase names. Eli merges this file with all other type-`sym` files and uses the resulting specification to create the procedure calls that build the tree.

pascal-.sym[86] ≡

```
{
 Equivalence classes of expression and operator phrases[72]
 Equivalence classes of statement phrases[78]
 }
```

This macro is attached to an output file.

7.10.4 type.c

A type-`c` file implements a module that characterizes a problem by solution. Eli does *not* merge type-`c` files, but compiles them individually.

type.c[87] ≡

```
{
 #include "scope.h"
 #include "type.h"
 #include "pdl_gen.h"
 #include "keyarray.h"

 void
 StandardIdProperties()
 {
 Set properties of standard names[55]
 }
 }
```

This macro is attached to an output file.

7.10.5 type.h

A type-`h` file defines the interface of a C module. Eli does not merge type-`h` files.

type.h[88] ≡

```
{
 #ifndef TYPE_H
 #define TYPE_H
 Kinds of objects[51]
 extern void StandardIdProperties();
 #endif
}
```

This macro is attached to an output file.

7.10.6 type.head

The interface of a C module is made available to the tree construction and attribute evaluation modules by placing a C pre-processor `include` directive in a type-`head` file. Eli merges the contents of all type-`head` files into a single file, `HEAD.h`, which is included by the tree construction and attribute evaluation modules.

type.head[89] ≡

```
{
 Definition table key array module interface[80]
 #include "type.h"
}
```

This macro is attached to an output file.

8 A Pascal Computer

The Pascal computer is an operational definition of the model of computation underlying Pascal-. It is an abstract machine with a memory, a *base register* capable of addressing that memory, and a processor. The processor has an internal stack, and is capable of executing a sequence of operations that evaluate expressions, transmit values between the processor stack and memory, transmit values to and from an external device, and control the sequence of execution.

Any program written in Pascal- can be expressed, without loss of information, as a program for the Pascal computer. This chapter describes the components of the model of computation defined by the Pascal computer, and shows how its operations are expressed as symbolic instructions.

8.1 Variable access

Variables in Pascal- exist only during an activation of the procedure in which they are declared. An *activation record* is defined for each procedure, and each time that procedure is invoked storage for a new instance of its activation record is allocated dynamically. When the procedure returns, the storage is freed. Each variable declared in a procedure becomes a field in the activation record for that procedure, and is accessed relative to the beginning of the activation record. Operations that access a variable must therefore specify two pieces of information: the address of the activation record holding that variable and the displacement of the variable's storage from the beginning of that activation record. The displacement of a variable's storage is known at compile time, because the compiler maps the variables declared in a procedure into fields in the activation record for that procedure. Because the activation records themselves are allocated dynamically, however, their addresses are not known until the program is executed.

The scope rules of Pascal- guarantee that the only variables that can be accessed directly are those in the currently-executing procedure and any procedures containing the declaration of the currently-executing procedure. By convention, the base register of the Pascal- computer always holds the address of the activation record of the currently-executing procedure. In addition to the fields representing variables of the procedure, each activation record has a field called the *static chain* that represents the address of the activation record for the procedure in which the currently-executing procedure was declared. Therefore the address of any activation record can be found by starting at the activation record addressed by the base register and following the static chain a specific number of steps. The number of steps can be determined by the compiler: if the variable is local to the current procedure, 0 steps are required; if it is local to the procedure containing the current procedure's definition, 1 step is required, and so forth.

Three operations are defined to place variable and parameter addresses onto the processor's stack:

Variable access[90] ≡
```
  {
   Variable:
     "Variable(" $/*Static chain steps*/ "," $/*Displacement*/ ")\n"

   ValParam:
     "ValParam(" $/*Static chain steps*/ "," $/*Displacement*/ ")\n"

   VarParam:
     "VarParam(" $/*Static chain steps*/ "," $/*Displacement*/ ")\n"
```
   Array element and record field access[91]
```
  }
```
This macro is invoked in definition 101.

`Variable` obtains the address of the local activation record from the base register, follows the static chain the specified number of steps to find the address of the activation record containing the desired variable, and adds the specified displacement. The result, which is the address of the variable, is pushed onto the processor's stack.

Parameters and variables are stored in different parts of the activation record, and therefore their displacements are interpreted differently. `ValParam` behaves like `Variable`, except that it interprets the displacement as a parameter displacement rather than a variable displacement.

The activation record field representing a variable parameter contains the address of the variable rather than its value. `VarParam` thus behaves like `ValParam` except that instead of pushing the sum of the activation record address and the displacement onto the stack it pushes the contents of the field addressed by that sum.

Brinch-Hansen does not provide a distinct `ValParam` operation in his description of the Pascal computer. Instead, he fixes the relationship between the parameter and variable storage areas of the activation record and uses `Variable` to access both. This tactic violates a basic rule of modularity: encapsulate design decisions that may change. A Pascal computer is an abstraction that must be realized on a variety of different real machines, and the layout of an activation record is something that depends strongly on the machine.

Addresses of array elements and record fields must be computed in two steps, because the array or record itself might be a variable parameter. The first step places the address of the array or record onto the stack, the second uses the appropriate displacement to compute the address of the element or field. (In the case of an indexed reference, the value of the subscript expression is computed and placed on the stack before the second step.)

Array element and record field access[91] ≡
```
{
 Index:
   $/*Indexed variable*/
   $/*Index expression*/
   "Index(" $/*Lower*/ "," $/*Upper*/ "," $/*Length*/ "," $/*Line*/ ")\n"

 Field:
   $/*Record variable*/
   "Field(" $/*Displacement*/ ")\n"
}
```
This macro is invoked in definition 90.

`Index` first removes the address of the array variable and the value of the subscript expression from the processor's stack. If the subscript value is not in bounds, `Index` reports an error at the specified source line. Otherwise it subtracts the value of the lower bound and multiplies by the length of an element to obtain the relative address of the element. This relative address is added to the address of the indexed variable and the result is pushed onto the processor's stack.

`Field` removes the address of the record variable from the processor's stack, adds the specified displacement, and pushes the result.

8.2 Expression evaluation

An expression may be a constant, the value of a variable, or a computation involving the values of other expressions. In each case, the process of expression evaluation leads to a value on the processor's stack.

Expression evaluation[92] ≡
```
{
 Constant:
   "Constant(" $/*Integer*/ ")\n"

 Value:
   $/*Variable address*/
   "Value(" $/*Number of memory locations*/ ")\n"

 Computation[95]
}
```
This macro is invoked in definition 101.

`Constant` simply pushes the specified value onto the stack. `Value` first removes the address of the variable from the stack and then pushes the specified number of memory locations, starting at that address.

There are two kinds of computation in Pascal-, those having one operand and those having two. In each case the operation removes the appropriate number of values from the processor's stack, uses them to compute a single result, and then pushes that result onto the processor's stack.

Monadic[93](◇2) ≡
```
{
 ◇1:
    $/*Expression*/
    "◇2\n"
  }
```
This macro is invoked in definitions 95 and 95.

Dyadic[94](◇2) ≡
```
  {
   ◇1:
    $/*Expression*/
    $/*Expression*/
    "◇2\n"
  }
```
This macro is invoked in definitions 95, 95, 95, 95, 95, 95, 95, 95, 95, 95, 95, 95, and 95.

Computation[95] ≡
```
  {
   Dyadic[94]('Lss','Less')
   Dyadic[94]('Leq','NotGreater')
   Dyadic[94]('Gtr','Greater')
   Dyadic[94]('Geq','NotLess')
   Dyadic[94]('Equ','Equal')
   Dyadic[94]('Neq','NotEqual')
   Nop:
    $/*Expression*/

   Dyadic[94]('Add','Add')
   Dyadic[94]('Sub','Subtract')
   Dyadic[94]('Mul','Multiply')
   Dyadic[94]('Div','Divide')
   Dyadic[94]('Mod','Modulo')
   Monadic[93]('Neg','Minus')

   Dyadic[94]('And','And')
   Dyadic[94]('Or','Or')
   Monadic[93]('Not','Not')
  }
```
This macro is invoked in definition 92.

8.3 Statement execution

The simplest statements are those that transmit information between the processor, the memory and the external device. They consist of single processor operations, whereas control statements consist of sequences of processor operations.

Statement execution[96] ≡
```
  {
   AssignmentStatement:
    $/*Variable address*/
    $/*Expression value*/
    "Assign(" $/*Length*/ ")\n"

   ReadStatement:
```

```
  $/*Variable address*/
  "Read\n"

WriteStatement:
  $/*Expression value*/
  "Write\n"
```

Control statements[97]
}

This macro is invoked in definition 101.

`AssignmentStatement` stores the contents of a specified number of elements from the processor stack at a specified location in memory. Both the elements and the address are removed from the stack. `ReadStatement` removes the variable address from the processor's stack and reads a single integer value from the external device into that address in memory. `WriteStatement` removes a single integer from the processor's stack and writes it to the external device.

Control statements use results obtained by the processor to control the execution sequence. Normally, operations are executed in the order in which they are given. Two operations, `Do` and `Goto`, are used to alter the normal order. Each of these operations nominates a specific operation to be executed next. They do so by giving a name. Names are associated with operations by the pseudo-operation `DefAddr`. `DefAddr` associates the specified name with the next operation in the sequence. A sequence of `DefAddr` pseudo-operations is allowed; in that case *all* of the names are associated with the operation following the sequence of pseudo-operations.

`Do` removes one element from the processor's stack. If that element's value is 0, then the next operation executed will be the operation whose name is specified by the `Do` operation. Otherwise the next operation executed will be the next in sequence.

The operation executed after a `Goto` operation will always be the operation whose name is specified by the `Goto` operation.

Control statements[97] ≡

```
{
 WhileStatement:
   "DefAddr(L" $1/*Label*/ ")\n"
   $2/*Expression*/
   "Do(L" $4/*Label*/ ")\n"
   $3/*Statement*/
   "Goto(L" $1 ")\n"
   "DefAddr(L" $4/*Label*/ ")\n"

 OneSided:
   $1/*Expression*/
   "Do(L" $3/*Label*/ ")\n"
   $2/*Statement*/
   "DefAddr(L" $3/*Label*/ ")\n"

 TwoSided:
   $1/*Expression*/
   "Do(L" $3/*Label*/ ")\n"
   $2/*Statement*/
   "Goto(L" $5 ")\n"
   "DefAddr(L" $3/*Label*/ ")\n"
   $4/*Statement*/
```

```
      "DefAddr(L" $5/*Label*/ ")\n"
    }
```
This macro is invoked in definition 96.

8.4 Procedure activation

Each procedure consists of a sequence of operations. Before executing these operations, a new activation record must be established for the procedure. The size of the parameter and variable storage areas in the activation record, and the amount of storage required by the procedure on the processor's stack are all known to the compiler. If the processor's stack does not have enough free storage, the `Procedure` operation terminates execution of the program and reports an error at the source program line containing the procedure definition.

In addition to allocating storage for the activation record, the field representing the static chain must be set to address the activation record of the procedure in which the new procedure was declared. This address is not known to the compiler, and must be supplied at run time by the `ProcCall` operation. The procedure being called must be visible to the calling procedure, because Pascal- only allows direct calls (there are no procedure-valued variables or parameters). Because the called procedure is visible, the activation record of the procedure in which the called procedure is defined can be found by stepping along the static chain, and the compiler can determine the number of steps required.

Before calling a procedure, the caller places the procedure's argument values onto the processor's stack in order from left to right. On return, the procedure deletes these values from the processor's stack.

Procedure activation[98] ≡
```
  {
   ProcedureDefinition:
     $6/*Nested procedure definitions*/
     "Procedure(" $1/*Parameter size*/ "," $2/*Local size*/
       "," $3/*Temp size*/ ",L" $4/*Label*/ "," $5/*Line*/ ")\n"
     $7/*Statements*/
     "EndProc(" $1/*Parameter size*/ ")\n"

   ProcedureStatement:
     $/*Arguments*/
     "ProcCall(" $/*Static chain steps*/ ",L" $/*Label*/ ")\n"

  }
```
This macro is invoked in definition 101.

Nested procedure definitions are separated in the Pascal- computer code because the only effect of procedure nesting in Pascal- is to provide scopes for variables. Once the variable accesses have been expressed in terms of activation records, the procedure nesting is redundant.

8.5 Program execution

A complete program acts like a procedure called by some external agency. It needs no name, however, because the operator that begins it is unique (`Program` rather than `Procedure`). Also, because a program has no arguments, program termination does not require anything to be removed from the processor's stack.

Program execution[99] ≡

```
{
 Program:
   "Program(" $/*AR size*/ "," $/*Temp size*/ "," $/*Line*/ ")\n"
    $/*Statements*/
    "EndProg\n"
 }
```
This macro is invoked in definition 101.

8.6 Code syntax

The fundamental relationship among the operations of the Pascal- computer is the sequence: one operation following another. Because execution begins with the program, its operations appear at the beginning of the program text.

In order to facilitate testing of the compiler, the program for the Pascal computer is considered to be a sequence of C pre-processor macro calls. Each operation is defined by a C pre-processor macro, and these definitions are provided by including them in the program text when it is provided to the C compiler.

Code syntax[100] ≡
```
{
 Sequence:
   $ $

 Text:
   "#include \"pascal.h\"\n\n"
   $/*Program text*/
   $/*Procedure bodies*/
 }
```
This macro is invoked in definition 101.

8.7 Specification file for symbolic machine code

Templates for producing structured output are specified in a type-ptg file.

computer.ptg[101] ≡

{
Variable access[90]
Expression evaluation[92]
Statement execution[96]
Procedure activation[98]
Program execution[99]
Code syntax[100]
}

This macro is attached to an output file.

9 Code Generation

The purpose of code generation is to translate the concepts of the source language into the concepts of the target machine. No errors are detected or reported during code generation.

Code generation is an attribution process. Given the information provided by type analysis, it determines the amount of memory required to store objects of each type, allocates storage to parameters and variables, and then produces a target implementation of the algorithm.

9.1 Operation parts

Every instruction output by the compiler consists of an operation part followed by zero or more operands. The operation parts are provided as literal strings by the templates described in Chapter 8. Each template has a name, and the code corresponding to the template is generated by invoking a function whose name is `PTG` followed by the template name. For example, code for the instruction `Variable(Level,Displ)` is generated by the call `PTGVariable(Level,Displ)`. The result of `PTGVariable(Level,Displ)` is the generated code in the form of a `PTGNode` value that can be used as an argument to other code generation functions.

The generated code is not actually emitted by the generation functions, as it is by Brinch-Hansen's `Emit` functions. Instead, an explicit data structure is constructed that contains the generated code. This data structure is a tree, and it is actually output by passing its root to the function `PTGOut`:

Operation parts[102] ≡
```
{
 SYMBOL Program COMPUTE PTGOut(THIS.Code) END;
}
```
This macro is invoked in definition 129.

9.2 Variable addressing

Variable access in the Pascal computer is described in Section 8.1: The address generated by the compiler specifies the number of steps that must be taken along the static chain to obtain the address of the proper activation record, and the relative address of the variable within that activation record. To compute these two components, the compiler must gather information from the program's declarations and store it in the definition table. An address can then be generated by combining information about the object with information determined from the context.

9.2.1 Static nesting level property

The compiler associates a *static nesting level* with the program and with each procedure. It is 1 for the program, 2 for each procedure declared in the program, 3 for each procedure declared in a level-2 procedure, and so forth. A `Contour` is is a phrase that has a static nesting level associated with it. Each variable is then given a property whose value is the static nesting level at which that variable was declared.

Static nesting level property[103] ≡
```
{
 Level: int;
}
```
This macro is invoked in definition 133.

Determination of the static nesting level[104] ≡
```
{
 ATTR Level: int;
 SYMBOL Contour COMPUTE INH.Level=ADD(INCLUDING Contour.Level,1); END;
SYMBOL Program INHERITS Contour COMPUTE INH.Level=1; END;
SYMBOL ProcedureBlock INHERITS Contour END;
}
```
This macro is invoked in definition 129.

The number of steps that must be taken along the static chain to obtain the address of the activation record containing a variable is the difference between the static nesting level of the procedure containing the reference and the static nesting level of the procedure in which the variable was declared.

Static chain steps to find the proper activation record address[105] ≡
```
{
 PTGNumb(SUB(INCLUDING Contour.Level,GetLevel(VariableNameUse.Key,0)))
}
```
This macro is invoked in definitions 115, 115, and 115.

There are two kinds of PTG functions: those that create data leaves and those that create nodes without data. Data can be stored only at data leaves, and a single data leaf can store an arbitrary number of data items of arbitrary types. The number of static chain steps to find the proper activation record address is an integer datum that must be stored at a data leaf, and `PTGNumb` is a library function that creates a data leaf holding a single integer value. This function is obtained by instantiating the generic library module `$/Tool/lib/Tech/LeafPtg.gnrc`.

9.2.2 Storage requirement property

In order to compute relative addresses within an activation record, the compiler needs to know the size of each variable. Brinch-Hansen, in his Algorithm 9.3, computes the size of each declared object at the point of declaration from the definition of its type. Thus the size of objects of a particular type is computed once for every object declared to be of that type. A better approach is to compute an appropriate size when a type is defined, and then use a property of the type to make that size available when the declaration of a new object of that type is encountered.

Storage requirement property[106] ≡
```
{
 Space: StorageRequired;
}
```
This macro is invoked in definition 133.

Because all symbols must be defined before they are used in Pascal-, storage requirements can be computed in textual order. The sizes of the standard types `Boolean` and `integer` are determined by the specification, while the sizes of user-defined types are determined by the compiler.

Determination of storage requirements[107] ≡
```
{
 CHAIN Storage: VOID;

 SYMBOL StandardBlock COMPUTE
   CHAINSTART HEAD.Storage=
     ORDER(
       SetSpace(IntegerKey,NewStorage(1,1,0),NoStorage),
       SetSpace(BooleanKey,NewStorage(1,1,0),NoStorage));
 END;
}
```

This macro is defined in definitions 107, 108, and 110.
This macro is invoked in definition 129.

In the Pascal computer, each value of standard type occupies one memory location.

An array type requires storage equal to the number of elements times the storage requirement of a single element.

Determination of storage requirements[108] ≡

```
{
 RULE ArrayDef: NewArrayType ::= 'array' '[' IndexRange ']' 'of' TypeNameUse
 COMPUTE
   NewArrayType.Storage=
     SetSpace(
       INCLUDING NewType.Key,
       ArrayStorage(
         ADD(SUB(IndexRange.UpperBound,IndexRange.LowerBound),1),
         GetSpace(TypeNameUse.Type,NoStorage)),
       NoStorage)
     DEPENDS_ON NewArrayType.Storage;
 END;
 }
```

This macro is defined in definitions 107, 108, and 110.
This macro is invoked in definition 129.

`ArrayStorage` is an operation exported by the data mapping module of the Eli library. It computes the storage requirements for an array of objects whose individual storage requirements are known. The first argument of `ArrayStorage` must be the number of elements in the array, and the second must be the storage requirements of the element type.

The data mapping module is an abstract data type, not a generic module. Therefore it need not be instantiated, but its interface must be made available:

Data mapping module interface[109] ≡

```
{
 #include "storage.h"
 }
```

This macro is invoked in definition 132.

Record types are more complex, because in addition to determining the total storage requirement of the record, the compiler must assign a relative address to each field. The compiler begins by establishing an empty storage area for the record. It then processes the field definitions in textual order, concatenating the storage required by each field to the record's storage area. The concatenation operation `Concatenate` is exported by the data mapping module. It takes two arguments: the storage requirements of the area to which the field must be added and the storage requirements of the field type. The function returns the address of the field relative to the beginning of the record, and updates the storage requirement of the area to reflect the addition of the field.

The relative address returned by `Concatenate` is stored as the `Displ` property of the field object (see the next section).

Determination of storage requirements[110] ≡

```
{
 SYMBOL NewRecordType: Area: StorageRequired;

 RULE RecordDef: NewRecordType ::= 'record' FieldList 'end'
COMPUTE
  NewRecordType.Area=NewStorage(0,1,0) DEPENDS_ON NewRecordType.Storage;
  NewRecordType.Storage=
    SetSpace(INCLUDING NewType.Key,NewRecordType.Area,NoStorage);
END;

SYMBOL FieldNameDef COMPUTE
  THIS.Storage=
    SetDispl(
      THIS.Key,
      Concatenate(INCLUDING NewRecordType.Area,INCLUDING Group.Space),
      0)
    DEPENDS_ON THIS.Storage;
END;
}
```
This macro is defined in definitions 107, 108, and 110.
This macro is invoked in definition 129.

Objects (variables, parameters, fields) are declared in groups in Pascal-: If several objects of the same type are being declared, the type is only stated once. This mechanism requires that the storage requirement of the type be made the value of an attribute of the phrase representing the group of objects, so that it will be available at all of the object declarations:

Distributing a storage requirement[111] ≡
```
{
 ATTR Space: StorageRequired;

 SYMBOL Group COMPUTE
   SYNT.Space=
     GetSpace(CONSTITUENT TypeNameUse.Type,NoStorage)
     DEPENDS_ON THIS.Storage;
 END;

 SYMBOL VariableDefinition INHERITS Group END;
 SYMBOL ParameterDefinition INHERITS Group END;
 SYMBOL RecordSection INHERITS Group END;
}
```
This macro is invoked in definition 129.

9.2.3 Relative address property

The relative address of an object within its storage area (a record for fields, an activation record for parameters and variables) is a property of that object.

Relative address property[112] ≡
```
{
 Displ: int;
}
```
This macro is invoked in definition 133.

A relative address is computed from an object's declaration, in the process of allocating space in a storage area. Allocation for field objects was specified in the last section, and allocation for objects in activation records is similar. The major difference is that an activation record has more structure than a record type.

Three kinds of information are stored in an activation record: parameters, local variables, and *overhead* (information like the static chain, needed to maintain the relationships among activation records). Parameter and local variable storage is determined by the declarations for the `Contour`, but storage for the overhead is determined by the target computer. Overhead information is maintained as a side effect of executing the instructions of the Pascal computer, and hence its size and placement depend on the implementation of those instructions. Brinch-Hansen, in Section 8.2 of his book, defines the overhead information to be three storage locations at the beginning of the storage allocated for variables. The first location is the static chain, holding the address of the activation record of the enclosing procedure, the second location is the *dynamic chain*, holding the contents of the base register during execution of the caller, and the third location is the *return address*, holding the location of the instruction at which execution should resume when the current procedure terminates. In the C implementation of the Pascal computer that accompanies this specification, there is no explicit return address. The third location of the overhead information is therefore free to hold the address of the parameter portion of the activation record.

Storage allocation[113] ≡
```
{
 SYMBOL Contour: Parameters, Variables: StorageRequired;
 SYMBOL Contour COMPUTE
   INH.Parameters=NewStorage(0,1,0);
   INH.Variables=NewStorage(3,1,0);
 END;
}
```
This macro is defined in definitions 113 and 114.
This macro is invoked in definition 129.

Parameter and variable objects have the `Displ` property, just like field objects, but they also have the `Level` property to record the static nesting level of the contour in which they are declared. Also, the storage requirement for a variable parameter is independent of its type because only the address of the parameter's value is stored. An address occupies one storage unit of the Pascal computer.

Storage allocation[114] ≡
```
{
 RULE VarParmDef:
   ParameterDefinition ::= 'var' ParameterNameDefList ':' TypeNameUse
 COMPUTE
   ParameterDefinition.Space=NewStorage(1,1,0);
 END;

 SYMBOL VariableNameDef COMPUTE
   THIS.Storage=
     ORDER(
       SetDispl(
         THIS.Key,
         Concatenate(INCLUDING Contour.Variables,INCLUDING Group.Space),
         0),
       SetLevel(THIS.Key,INCLUDING Contour.Level,0))
     DEPENDS_ON THIS.Storage;
 END;

 SYMBOL ParameterNameDef COMPUTE
   THIS.Storage=
     ORDER(
        SetDispl(
          THIS.Key,
          Concatenate(INCLUDING Contour.Parameters,INCLUDING Group.Space),
          0),
        SetLevel(THIS.Key,INCLUDING Contour.Level,0))
      DEPENDS_ON THIS.Storage;
  END;
  }
```

This macro is defined in definitions 113 and 114.
This macro is invoked in definition 129.

9.2.4 Accessing an object

Object access code pushes the address of the desired object onto the processor's stack in the Pascal computer. Access to an object of one of the standard types is specified in Pascal- by giving the name of the object, while access to an object of a user-defined type may involve subscripting or field selection. In that case the address of the object (array or record) containing the desired object is placed on the stack and then the appropriate selection operations are executed.

Section 8.1 pointed out the differences among the accessing operations for variables, value parameters and variable parameters. In addition, as noted in Section 2.3.2, a `VariableNameUse` may actually be a constant name. Since constants are not stored in memory, the code generated for a constant name must push the constant itself, not the address of the constant, onto the processor's stack.

Accessing an object[115] ≡

```
  {
  RULE VarIdn: VariableAccess ::= VariableNameUse
  COMPUTE
    VariableAccess.Code=
      IF(EQ(VariableAccess.Kind,Constantx),
        PTGConstant(PTGNumb(GetValue(VariableNameUse.Key,0))),
      IF(EQ(VariableAccess.Kind,ValueParameter),
        PTGValParam(
          Static chain steps to find the proper activation record address[105],
          PTGNumb(GetDispl(VariableNameUse.Key,0))),
      IF(EQ(VariableAccess.Kind,VarParameter),
        PTGVarParam(
          Static chain steps to find the proper activation record address[105],
          PTGNumb(GetDispl(VariableNameUse.Key,0))),
      PTGVariable(
        Static chain steps to find the proper activation record address[105],
        PTGNumb(GetDispl(VariableNameUse.Key,0))))));
  END;

  RULE Index: VariableAccess ::= VariableAccess '[' Expression ']'
  COMPUTE
    VariableAccess[1].Code=
      PTGIndex(
        VariableAccess[2].Code,
        Expression.Code,
        PTGNumb(GetLowerBound(VariableAccess[2].Type,0)),
        PTGNumb(GetUpperBound(VariableAccess[2].Type,0)),
        PTGNumb(
        StorageSize(
          GetSpace(GetElementType(VariableAccess[2].Type,NoKey),NoStorage))),
      PTGNumb(LINE));
END;

RULE Select: VariableAccess ::= VariableAccess '.' FieldNameUse
COMPUTE
  VariableAccess[1].Code=
    PTGField(
      VariableAccess[2].Code,
      PTGNumb(GetDispl(FieldNameUse.Key,0)));
END;
}
```

This macro is invoked in definition 129.

9.3 Expression code

Code for an expression always leaves the value of the expression at the top of the processor's stack. If the expression is a `Numeral` or a `VariableAccess` then that value is obtained from either an operation or memory, respectively. Because a `VariableAccess` may actually be a reference to a constant, the compiler must check its `Kind` to determine whether the value at the top of the processor's stack is a value or an address at which a value can be found. When the top element of the processor's stack is an address, the compiler must emit a `Value` operation to obtain the contents of that address if the context demands a value.

Expression code[116] ≡

```
{
RULE Denotation: Expression ::= Numeral
COMPUTE
  Expression.Code=PTGConstant(PTGNumb(Numeral.Value));
END;

RULE VariableExpr: Expression ::= VariableAccess
COMPUTE
  Expression.Code=
    IF(OR(Expression.ExpectedVar,EQ(VariableAccess.Kind,Constantx)),
      VariableAccess.Code,
      PTGValue(
        VariableAccess.Code,
        PTGNumb(StorageSize(GetSpace(VariableAccess.Type,NoStorage)))));
END;
}
```

This macro is defined in definitions 116, 117, and 120.
This macro is invoked in definition 129.

If the expression is neither a `Numeral` nor a `VariableAccess` then its value is obtained by applying the appropriate operator to one or two operands. The code generation process is independent of the particular operator, which is supplied by the operator child of the expression, although it does depend on the number of operands. LIDO does not allow an attribute to be applied as a function, however, so a circumlocution is necessary:

Expression code[117] ≡

```
{
 ATTR Op: PtgFunc;

 RULE Monadic: Expression ::= Unop Expression
 COMPUTE
   Expression[1].Code=Apply1(Unop.Op,Expression[2].Code);
 END;

 RULE Dyadic: Expression ::= Expression Binop Expression
 COMPUTE
   Expression[1].Code=Apply2(Binop.Op,Expression[2].Code,Expression[3].Code);
 END;
 }
```

This macro is defined in definitions 116, 117, and 120.
This macro is invoked in definition 129.

`PtgFunc`, `Apply1` and `Apply2` are defined in C:

PTG functions as attributes[118] ≡

```
{
 typedef PTGNode (*PtgFunc)();
 #define Apply1(f,x) ((*f)(x))
 #define Apply2(f,x,y) ((*f)(x,y))
 }
```

This macro is invoked in definition 132.

These definitions say that `PtgFunc` objects are functions that return PTG nodes, `Apply1` applies its first argument to its second, and `Apply2` applies its first argument to its second and third.

All of the operator rules have exactly the same form, defining the `Op` attribute of the left-hand side as the appropriate PTG function:

Operator encoding[119](◇3) ≡

```
{
 RULE r◇1: ◇2 ::= ◇3 COMPUTE ◇2.Op=PTG◇1 END;
 }
```

This macro is invoked in definitions 120, 120, 120, 120, 120, 120, 120, 120, 120, 120, 120, 120, 120, 120, 120, and 120.

Expression code[120] ≡

{
 Operator encoding[119](‘`Lss`’,‘`Binop`’,‘`'<'`’)
 Operator encoding[119](‘`Leq`’,‘`Binop`’,‘`'<='`’)
 Operator encoding[119](‘`Gtr`’,‘`Binop`’,‘`'>'`’)
 Operator encoding[119](‘`Geq`’,‘`Binop`’,‘`'>='`’)
 Operator encoding[119](‘`Equ`’,‘`Binop`’,‘`'='`’)
 Operator encoding[119](‘`Neq`’,‘`Binop`’,‘`'<>'`’)

 Operator encoding[119](‘`Add`’,‘`Binop`’,‘`'+'`’)
 Operator encoding[119](‘`Sub`’,‘`Binop`’,‘`'-'`’)
 Operator encoding[119](‘`Mul`’,‘`Binop`’,‘`'*'`’)
 Operator encoding[119](‘`Div`’,‘`Binop`’,‘`'div'`’)
 Operator encoding[119](‘`Mod`’,‘`Binop`’,‘`'mod'`’)

 Operator encoding[119](‘`Neg`’,‘`Unop`’,‘`'-'`’)

Operator encoding[119](`Nop`,`Unop`,`''+''`)

Operator encoding[119](`And`,`Binop`,`''and''`)
Operator encoding[119](`Or`,`Binop`,`''or''`)
Operator encoding[119](`Not`,`Unop`,`''not''`)
}

This macro is defined in definitions 116, 117, and 120.
This macro is invoked in definition 129.

9.4 Statement code

A `Statement` is a sequence of operations, which may be empty. The processor's stack is always empty before the operations of a statement are executed and after their execution is complete. Thus a statement may affect the state of the memory or external devices, but it neither expects nor yields a value. For example, the assignment statement consists of a sequence of operations that push the address of the variable to be assigned onto the processor's stack, push the value to be assigned, and then carry out the assignment removing both elements from the processor's stack.

Statement code[121] ≡

```
{
 RULE Empty: Statement ::=
 COMPUTE
   Statement.Code=PTGNULL;
 END;

 RULE Assign: Statement ::=  VariableAccess ':=' Expression
 COMPUTE
   Statement.Code=
     PTGAssignmentStatement(
       VariableAccess.Code,
       Expression.Code,
       PTGNumb(StorageSize(GetSpace(VariableAccess.Type,NoStorage))));
 END;

 RULE Compound: Statement ::=  CompoundStatement
 COMPUTE
   Statement.Code=CompoundStatement.Code;
 END;

 SYMBOL CompoundStatement COMPUTE
   SYNT.Code=
     CONSTITUENTS Statement.Code
       SHIELD Statement
       WITH (PTGNode, PTGSequence, IDENTICAL, PTGNull);
 END;
}
```

This macro is defined in definitions 121 and 123.
This macro is invoked in definition 129.

The `CONSTITUENTS` construct gathers all of the `Code` attributes of component `Statement` nodes, using `PTGSequence` to combine them pairwise. Note, however, that many `Statement` nodes are actually the roots of subtrees that contain `Statement` nodes themselves. The `SHIELD` clause prevents the

`CONSTITUENTS` construct from gathering the `Code` attributes of the `Statement` nodes within such subtrees.

Statements that alter the normal sequence of operations must be able to nominate a successor to the current operation. Section 8.3 described how operations can be named via the `DefAddr` pseudo-operation. The compiler must generate these names; it does so by obtaining a sequence of unique integers from the `NewInteger` module:

Unique integers[122] ≡

```
{
 static int Next = 1;     /* Next integer to be delivered */

 int NewInteger() { return Next++; }
}
```

This macro is invoked in definition 130.

The number of labels needed by the statement depends on its internal structure, described in Section 8.3.

Statement code[123] ≡

```
{
 RULE While: Statement ::=  'while' Expression 'do' Statement
 COMPUTE
   Statement[1].Code=
     PTGWhileStatement(
       PTGNumb(NewInteger()),
       Expression.Code,
       Statement[2].Code,
       PTGNumb(NewInteger()));
 END;

 RULE OneSided:  Statement ::=  'if' Expression 'then' Statement
 COMPUTE
   Statement[1].Code=
     PTGOneSided(
       Expression.Code,
       Statement[2].Code,
       PTGNumb(NewInteger()));
 END;

 RULE TwoSided: Statement ::=  'if' Expression 'then' Statement 'else' Statement
 COMPUTE
   Statement[1].Code=
     PTGTwoSided(
       Expression.Code,
       Statement[2].Code,
       PTGNumb(NewInteger()),
       Statement[3].Code,
       PTGNumb(NewInteger()));
 END;
}
```

This macro is defined in definitions 121 and 123.
This macro is invoked in definition 129.

9.5 Procedure code

Most of the code for a procedure definition is generated by its children. The storage requirements must be summarized and a label generated so that invocations can refer to the procedure.

Procedure code[124] ≡

```
{
 ATTR Label: int;

 RULE ProcDef:
   ProcedureDefinition ::= 'procedure' ProcedureNameDef ProcedureBlock ';'
 COMPUTE
   .Label=NewInteger();
   ProcedureDefinition.Code=
     PTGProcedureDefinition(
       PTGNumb(StorageSize(ProcedureBlock.Parameters)),
       PTGNumb(StorageSize(ProcedureBlock.Variables)),
       PTGNumb(0),        /*Temp size*/
       PTGNumb(.Label),
       PTGNumb(LINE),
       CONSTITUENTS ProcedureDefinition.Code
         SHIELD ProcedureDefinition
         WITH (PTGNode, PTGSequence, IDENTICAL, PTGNull),
       CONSTITUENTS CompoundStatement.Code
         SHIELD (ProcedureDefinition, CompoundStatement)
         WITH (PTGNode, PTGSequence, IDENTICAL, PTGNull))
     DEPENDS_ON ProcedureBlock.Storage;
   ProcedureBlock.Storage=
     ORDER(
       SetLevel(ProcedureNameDef.Key,ProcedureBlock.Level,0),
       SetLabel(ProcedureNameDef.Key,.Label,0))
     DEPENDS_ON ProcedureDefinition.Storage;
 END;
}
```

This macro is defined in definitions 124 and 127.
This macro is invoked in definition 129.

The `SHIELD` clauses in this rule avoid duplication of code: `CONSTITUENTS CompoundStatement.Code` would gather the bodies of nested procedures if it could penetrate the subtrees rooted in `ProcedureDefinition` nodes.

In addition to the `Level` property that it shares with other objects, a procedure name must have the procedure's label as a property.

Label property[125] ≡

```
{
 Label: int;
}
```

This macro is invoked in definition 133.

The label property is accessed at the procedure call.

Read and write statements in Pascal- have the form of procedure calls, but they are implemented by distinct operations of the Pascal computer. They must therefore be recognized specially during code generation.

When a user-defined procedure is invoked, it must be provided with the address of the activation record of the procedure in which it was declared. That address can be found by stepping along the static chain, exactly as in the case of variable addressing. There is a small difference, however: The number of steps is one larger than the difference between the current static nesting level and the static nesting level of the procedure being called. The reason is that that address sought is the activation record address for the procedure in which the called procedure was declared; the called procedure itself has no activation record until the `ProcCall` operation (Section 8.4) has been executed.

Steps to find the called procedure's environment[126] ≡

```
{
 PTGNumb(
   ADD(
     SUB(INCLUDING Contour.Level,GetLevel(ProcedureNameUse.Key,0)),
     1))
 }
```

This macro is invoked in definition 127.

Procedure code[127] ≡

```
{
 RULE Call: Statement ::=  ProcedureNameUse ActualParameterList
 COMPUTE
   .Code=CONSTITUENTS Expression.Code SHIELD Expression
     WITH (PTGNode, PTGSequence, IDENTICAL, PTGNull);
   Statement.Code=
     IF(EQ(ProcedureNameUse.Key,ReadKey),
       PTGReadStatement(.Code),
     IF(EQ(ProcedureNameUse.Key,WriteKey),
       PTGWriteStatement(.Code),
     PTGProcedureStatement(
       .Code,
       Steps to find the called procedure's environment[126],
       PTGNumb(GetLabel(ProcedureNameUse.Key,0)))))
     DEPENDS_ON Statement.Storage;
 END;
 }
```

This macro is defined in definitions 124 and 127.
This macro is invoked in definition 129.

The `SHIELD` clause forces the `CONSTITUENTS` construct to gather the `Code` attributes only from the argument expressions, and not from any of their components.

9.6 Program code

The code for the program definition is almost identical to the code for a procedure definition. There are no parameters for the program, and since the program is not invoked there is no need for a `Label` property.

Program code[128] ≡

```
{
 RULE Source: Program ::=  'program' ProgramName ';' BlockBody '.'
 COMPUTE
   Program.Code=
    PTGText(
      PTGProgram(
        PTGNumb(StorageSize(Program.Variables)),
        PTGNumb(0),  /*Temp size*/
        PTGNumb(LINE),
        CONSTITUENTS CompoundStatement.Code
          SHIELD (ProcedureDefinition,CompoundStatement)
          WITH (PTGNode, PTGSequence, IDENTICAL, PTGNull)),
        CONSTITUENTS ProcedureDefinition.Code
          SHIELD ProcedureDefinition
          WITH (PTGNode, PTGSequence, IDENTICAL, PTGNull));
END;
}
```
This macro is invoked in definition 129.

The `SHIELD` clauses prevent duplicate code: `CONSTITUENTS CompoundStatement.Code` would gather the bodies of all nested procedures as well as the body of the program if it were allowed to penetrate the subtrees rooted in `ProcedureDefinition` nodes.

9.7 Specification files for code generation

Five kinds of specifications are needed to define the Pascal- code generation problem to Eli.

9.7.1 code.lido

A type-`lido` file describes the attribution needed to gather the information and relate the generated code fragments. Eli will merge these attributions with others specified in this document and create a tree traversal algorithm.

code.lido[129] ≡
```
{
 ATTR Code: PTGNode;
```
Operation parts[102]
Determination of the static nesting level[104]
Determination of storage requirements[107]
Distributing a storage requirement[111]
Storage allocation[113]
Accessing an object[115]
Expression code[116]
Statement code[121]
Procedure code[124]
Program code[128]
}

This macro is attached to an output file.

9.7.2 code.c

A type-`c` file describes a problem by giving its solution. In the case of the code generation, the problem was that of generating unique labels.

code.c[130] ≡

{
#include "code.h"
Unique integers[122]
}

This macro is attached to an output file.

9.7.3 code.h

A type-h file gives the interface for the module described by the type-c file. By convention, an interface file is included if it is needed explicitly. This can lead to multiple inclusions, so every interface file must protect itself against this possibility as indicated. Conventionally, the symbol used is the upper-case version of the file name, with "." replaced by "_".

code.h[131] ≡
```
{
#ifndef CODE_H
#define CODE_H
extern int NewInteger();
#endif
}
```
This macro is attached to an output file.

9.7.4 code.head

A type-head file places CPP directives into the generated attribution routine. The specification of the PTG functions as attributes needs the definition of PTGNode, which is given by ptg_gen.h.

code.head[132] ≡

{
#include "code.h"
Data mapping module interface[109]
#include "ptg_gen.h"
PTG functions as attributes[118]
}

This macro is attached to an output file.

9.7.5 code.pdl

A type-pdl file defines properties. If any of the properties are objects whose types are not basic data types of C, the type-pdl file must contain a string naming an interface file where the properties' types are defined. Some of the properties defined here are of type StorageRequirement, which is defined in the file storage.h.

code.pdl[133] ≡

{
Static nesting level property[103]
Relative address property[112]
"storage.h"
Storage requirement property[106]
Label property[125]
}

This macro is attached to an output file.