An Analyzer for Java

W. M. Waite
W. E. Munsil

September 11, 2008

The purpose of this specification is to verify that the type analysis modules of Eli 4.5 are useful for describing a modern object-oriented language. It implements the typing rules of Java 1.4, and accepts the entire language. Many programming errors that do not involve type analysis are not reported, due to a lack of time and the particular goal of the project.

Complete name analysis for Java 1.4 requires a fixpoint algorithm for resolving type inheritance, and this specification does not include that algorithm. As a result, some correct programs will be rejected with reports of improper inheritance or undefined identifiers. This deficiency is not relevant for the type analysis demonstration.

Chapter 1 defines the building blocks of Java text, and Chapter 2 defines the programs that can be built from them. Together, these definitions provide the *phrase structure* of a Java program.

Generally speaking, the phrase structure of a program is not a convenient representation for analysis. Chapter 3 defines the *abstract syntax tree* (AST), a data structure that reflects the semantics of a Java program. Eli deduces most of the transformations required to obtain the AST from the phrase structure, but requires additional specification for some aspects (Section 3.3).

The computations specified in Chapters 4 and 5 decorate the nodes of the AST with information about the binding of identifiers and the type of expressions, respectively. Chapter 6 uses these decorations to report semantic errors.

We assume that Java packages are stored in a file system, and Appendix A defines how those packages are accessed. All classes needed for a compilation are assumed to be available as source text, and are added to the original text as individual compilation units. The entire assemblage is then processed as a unit.

Appendix B provides some additional information to aid in interpreting definition table keys when examining attributes with Noosa.

The first version of this specification was developed in 1996 to test whether Eli could deal with multiple-file input based on partial analysis of the text. That was successful, although the present specification uses a less fine-grained mechanism.

In 1998 the specification was extended to the full Java 1.0 language and used to implement some extensions for a Ph.D. thesis (Munsil, Wesley E. “Intensive Inheritance with Applications to Java”, Department of Computer Science, University of Colorado at Colorado Springs, 1998).

The Java 1.0 specification was enhanced to accept programs written in Java 1.4 in 2008. All of the type analysis computations were rewritten to use Eli 4.5’s type analysis modules, and Java Names were disambiguated in the abstract syntax tree by using a tree parser.

Eli 4.5 can generate an executable analyzer from the specifications used to derive this document.

Contents

Chapter 1

Lexical Structure

To solve the lexical analysis subproblem, a Java processor must examine each character of the input text, recognizing character sequences as tokens, comments or white space. Regular expressions are used to classify these sequences.

Once a character sequence has been classified, the sequence defined by the regular expression may be extended or shortened by an *auxiliary scanner*. An auxiliary scanner is associated with a regular expression by specifying it's name, enclosed in parentheses (e.g. `(auxNewLine)`).

Identifiers and denotations must be retained for further processing. This is done in a uniform way by retaining one copy of each distinct input string appearing as an identifier or a denotation. Each identifier or denotation is represented internally by the index of its string in the string memory. If `i` is this index, then `StringTable(i)` is a pointer to the (null-terminated) string.

A *token processor* can be associated with each regular expression by specifying its name, enclosed in brackets (e.g. `[mkidn]`). The token processor is a C routine, the intent of which is to construct an integer-valued internal representation of the scanned string. Every token processor obeys the following interface:

Token processor[1](◇1):

```
void
#if PROTO_OK
◇1(char *c, int l, int *t, int *s)
#else
◇1(c, l, t, s) char *c; int l, *t, *s;
#endif
/* On entry-
 *   c points to the first character of the scanned string
 *   l=length of the scanned string
 *   *t=initial classification
 * On exit-
 *   *t=final classification
 *   *s=internal representation
 ***/
```

This macro is invoked in definitions 10, 18, 25, and 31.

A type-`gla` file specifies the lexical analysis subproblem:

Phrase.gla[2]:

InputElement[5]

This macro is attached to a product file.

1.1 Unicode

This document does not describe the handling of Unicode input. A source program is assumed to consist solely of ASCII characters, and contain no Unicode escapes. The issue of Unicode is orthogonal to the remainder of the specification, affecting only the input routine, the specification of the `Identifier` token, and the implementation of the lexical analyzer.

1.2 Lexical Translation

1.3 Unicode Escapes

This provides a regular expression definition for Unicode escapes. This is not the correct (or general) way to handle Unicode escapes. It simply makes it possible to accept Unicode escapes inside character and string literals.

Unicode Escape[3]:

\\u*H*[4]*H*[4]*H*[4]*H*[4]

This macro is invoked in definitions 30 and 32.

H[4]:

```
[0-9a-fA-F]
```

This macro is invoked in definition 3.

1.4 Line Terminators

Eli normally recognizes the ASCII LF, CR, and CRLF as input line terminators.

1.5 Input Elements and Tokens

Eli does not separate the specification of tokens from the specification of the language structure, so there is no specification here of `Input` or `InputElements`. The definition of `InputElement` given here simply collects all of the regular expressions defining those elements:

InputElement[5]:

WhiteSpace[7]
Comment[8]
Token[6]

This macro is invoked in definition 2.

White space is defined in the next section.

Token[6]:

Keyword[9]
Identifier[11]
Literal[12]

This macro is invoked in definition 5.

Separators and operators do not have individual lexical descriptions in an Eli specification. They appear as literal terminals in the grammar, and their lexical descriptions are extracted from the grammar by Eli.

1.6 WhiteSpace

By default, Eli considers SP and HT characters to be white space. The FF character, however, must be added:

WhiteSpace[7]:

```
$\f
```

This macro is invoked in definition 5.

1.7 Comments

The three kinds of comment are specified by regular expressions and auxiliary scanners rather than a context-free grammar:

Comment[8]:

```
$"/*"   (auxCComment)
$"//"   (auxEOL)
```

This macro is invoked in definition 5.

The auxiliary scanner `auxCComment` accumulates characters up to and including the next occurrence of the sequence `*/`; `auxNewLine` was discussed above.

1.8 Keywords

Most of the Java keywords appear as literals in the grammar, and therefore do not need additional specifications. The keywords `const` and `goto` are reserved by Java, even though they are not currently used. This allows the processor to give a very specific error report if these C++ keywords are incorrectly used in Java programs.

Keyword[9]:

```
$const          [KeyErr]
$goto           [KeyErr]
```

This macro is invoked in definition 6.

Here the token processor `KeyErr` makes the error report:

Report a keyword error[10]:

```
Token processor[1](‘KeyErr’)
{ message(ERROR, "Illegal keyword -- ignored", 0, &curpos); }
```

This macro is invoked in definition 37.

When scanning of a character sequence begins, the scanner sets the variable `curpos` (exported by the error module) to the coordinates of the first character of the sequence. By using the address of this variable in the `message` call, `KeyErr` places the report at the start of the incorrect keyword.

1.9 Identifiers

This definition reflects the assumption discussed earlier that the input text would be ASCII rather than Unicode. A change to Unicode requires modification of the sets given here.

Identifier[11]:

```
Identifier:     $[A-Za-z$_][A-Za-z0-9$_]*       [mkidn]
```

This macro is invoked in definition 6.

The token processor `mkidn` encodes the string matched by the regular expression with a unique integer value. Every occurrence of a given string will be encoded with the same integer. A pointer to the characters of the string can be obtained by applying `StringTable` to the integer encoding.

1.10 Literals

A *literal* is the source code representation of a value of a primitive type or the `String` type or the null type. All literals are represented internally by integers. The literal string represented by the integer `i` is the value of `StringTable(i)`. Every instance of a particular literal is represented by the same integer.

Literal[12]:

```
Integer Literal[16]
Floating-Point Literal[20]
Character Literal[30]
String Literal[32]
```

This macro is invoked in definition 6.

The Boolean and null literals appear in the grammar as keywords and therefore have no separate lexical definition.

Numeric literals are converted to standard normalized forms and checked for validity before their integer representations are determined. This conversion is performed by Eli’s string arithmetic module, which is also used to evaluate constant expressions:

Phrase.specs[13]:

```
$/Tech/strmath.specs
```
Instantiate necessary modules[54]

This macro is attached to a product file.

By default, Eli's string arithmetic module will represent a value whose exponent is `-1` with a leading `0`. This avoids the need for an explicit exponent, but does not conform to the definition of a normalized number. We therefore instruct the module not to perform such "de-normalization":

Phrase.head[14]:

```
#include "strmath.h"
```

This macro is attached to a product file.

Phrase.init[15]:

```
(void)strmath(STRM_DENORMALIZE, 0);
```

This macro is attached to a product file.

1.10.1 Integer Literals

Integer literals are decomposed by type in order to preserve type information gleaned during lexical analysis. If this information were not preserved, later components of the compiler would be forced to re-scan the characters of the literal to obtain it.

Integer Literal[16]:

```
IntLiteral:     $0|[1-9][0-9]*          [mkint10]
                $0[xX][0-9a-fA-F]+      [mkint16]
                $0[0-7]+                [mkint8]
LongLiteral:    $(0|[1-9][0-9]*)[lL]    [mklng10]
                $0[xX][0-9a-fA-F]+[lL]  [mklng16]
                $0[0-7]+[lL]            [mklng8]
```

This macro is invoked in definition 12.

The grammar symbol `IntegerLiteral` is expanded to retain the type information:

IntegerLiteral[17]:

```
IntLiteral / LongLiteral
```

This macro is invoked in definition 39.

The token processors normalize the integer to decimal form to guarantee that integer literals with the same value have the same internal representation. Thus if `i` is the internal representation of an integer literal, `StringTable(i)` is always a null-terminated sequence of decimal digits from the set `[0-9]`.

Normalize a literal value[18]:

Token processor[1](‘mkint10’)
```
{ NormInt(c, l, t, s, 10); *t = IntLiteral; }
```

Token processor[1](‘mkint16’)
```
{ NormInt(c + 2, l - 2, t, s, 16); *t = IntLiteral; }
```

Token processor[1](‘mkint8’)
```
{ NormInt(c, l, t, s, 8); *t = IntLiteral; }
```

Token processor[1](‘mklng10’)
```
{ NormInt(c, l - 1, t, s, 10); *t = LongLiteral; }
```

Token processor[1](‘mklng16’)
```
{ NormInt(c + 2, l - 3, t, s, 16); *t = LongLiteral; }
```

Token processor[1](‘mklng8’)
```
{ NormInt(c, l - 1, t, s, 8); *t = LongLiteral; }
```

This macro is defined in definitions 18, 25, and 31.
This macro is invoked in definition 37.

The actual normalization is performed by invoking the string arithmetic package from the Eli library:

*NormInt(char *c, int l, int *t, int *s, int r)*[19]:

```
/* On entry-
 *   c points to the first character of the scanned string
 *   l=length of the scanned string
 *   *t=initial classification
 *   r=radix of the number to be normalized
 * On exit-
 *   *t=final classification
 *   *s=internal representation
 ***/
{ char save, *num, *temp, complement[ARITH_SIZE*2];
  int error;

  save = c[l]; c[l] = '\0';

  /* Maximum values taken from section 3.10.1 of the language specification */
  if (*t == IntLiteral) {
    if (r == 8) {
      temp = strsub(c, "40000000000", 8);
      error = (!temp || temp[0] != '-');
      if (!error) {
        strcpy(complement, temp);
        num = strsub("17777777777", c, 8);
        if (num[0] == '-')
          num = strnorm(complement, 8, 10, "");
        else
```

```
            num = strnorm(c, 8, 10, "");
          }
        } else if (r == 10) {
          temp = strsub(c, "2147483649", 10);
          error = (!temp || temp[0] != '-');
          if (!error)
            num = strnorm(c, 10, 10, "");
        } else {    /* r == 16 */
          temp = strsub(c, "100000000", 16);
          error = (!temp || temp[0] != '-');
          if (!error) {
            strcpy(complement, temp);
            num = strsub("7fffffff", c, 16);
            if (num[0] == '-')
              num = strnorm(complement, 16, 10, "");
            else
              num = strnorm(c, 16, 10, "");
          }
        }
      } else {        /* *t == LongLiteral */
        if (r == 8) {
          temp = strsub(c, "2000000000000000000000", 8);
          error = (!temp || temp[0] != '-');
          if (!error) {
            strcpy(complement, temp);
            num = strsub("7777777777777777777777", c, 8);
            if (num[0] == '-')
              num = strnorm(complement, 8, 10, "");
            else
              num = strnorm(c, 8, 10, "");
          }
        } else if (r == 10) {
          temp = strsub(c, "9223372036854775809", 10);
          error = (!temp || temp[0] != '-');
          if (!error)
            num = strnorm(c, 10, 10, "");
        } else {    /* r == 16 */
          temp = strsub(c, "10000000000000000", 16);
          error = (!temp || temp[0] != '-');
          if (!error) {
            strcpy(complement, temp);
            num = strsub("7fffffffffffffff", c, 16);
            if (num[0] == '-')
              num = strnorm(complement, 16, 10, "");
            else
              num = strnorm(c, 16, 10, "");
          }
        }
      }

      if (error) {
```

```
    message(ERROR, "Integer overflow", 0, &curpos);
    *s = 0; return;
  }

  c[l] = save;
  mkidn(num, strlen(num), t, s);
}
```

This macro is invoked in definition 37.

If an overflow is detected, the internal representation is set to `0`. `StringTable[0]` is the internal representation of a null string, and is therefore different from the representation of any valid integer.

1.10.2 Floating-Point Literals

Floating-point literals are decomposed by type in order to preserve type information gleaned during lexical analysis. If this information were not preserved, later components of the compiler would be forced to re-scan the characters of the literal to obtain it.

A floating-point literal has the following parts: a whole-number part, a decimal point, a fractional part, an exponent, and a type suffix. The exponent, if present, is indicated by the letter `e` or `E` followed by an optionally signed integer.

At least one digit, in either the whole number or the fractional part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional.

Floating-Point Literal[20]:

```
DoubleLiteral:  $((N[22]E[23]?|D[21]+E[23])[dD]?|D[21]+E[23]?[dD])  [mkflt]
FloatLiteral:   $(N[22]E[23]?[fF]?|D[21]+(E[23][fF]?|[fF]))        [mkflt]
```

This macro is invoked in definition 12.

The `D` macro represents a single digit:

D[21]:

```
[0-9]
```

This macro is invoked in definitions 20, 22, and 23.

The `N` macro represents a sequence of digits with a leading, trailing or embedded decimal point:

N[22]:

```
(D[21]+\.D[21]*|\.D[21]+)
```

This macro is invoked in definition 20.

The `E` macro represents an exponent:

E[23]:

```
([eE][+-]?D[21]+)
```

This macro is invoked in definition 20.

The grammar symbol `FloatingPointLiteral` is expanded to retain the type information:

FloatingPointLiteral[24]:

```
FloatLiteral / DoubleLiteral
```

This macro is invoked in definition 39.

The token processor `mkflt` converts the value to IEEE 754 format, expressed as a character string. Thus if `i` is the internal representation of a floating point literal, `StringTable(i)` is always in normalized IEEE 754 format. Digits of single-precision values are binary, those of double-precision values are hexadecimal. Exponents are always sequences of decimal digits.

Normalize a literal value[25]:

```
Token processor[1](‘mkflt’)
{ char save, *temp; int DomainError;

  /* If there is a type specifier on the end, remove it since we don’t */
  /* need it anymore. */
  if (*t == FloatLiteral || c[l-1] == ’d’ || c[l-1] == ’D’) l--;

  (void)strmath(STRM_EXP_SYMBOLS, "eE");
  (void)strmath(STRM_INTEGER_SIZE, 1);
  (void)strmath(STRM_ROUND_SIZE, ARITH_SIZE-1);
  save = c[l]; c[l] = ’\0’;
  Convert a Floating-Point Literal[26]
  if (*t == FloatLiteral) {
    Check for over- or underflow in a FloatLiteral[27]
  } else {
    Check for over- or underflow in a DoubleLiteral[28]
  }
  c[l] = save;
  (void)strmath(STRM_ROUND_SIZE, 0);
  (void)strmath(STRM_INTEGER_SIZE, ARITH_SIZE);

  if (!CsmStrPtr || DomainError) {
    message(ERROR, "Value out of range", 0, &curpos);
    obstack_free(Csm_obstk, CsmStrPtr); *s = 0; return;
  }

  mkidn(CsmStrPtr, strlen(CsmStrPtr), t, s);
}
```

This macro is defined in definitions 18, 25, and 31.
This macro is invoked in definition 37.

If an overflow is detected, the internal representation is set to `0`. `StringTable[0]` is the internal representation of a null string, and is therefore different from the representation of any valid number.

A `FloatingPointLiteral` is regarded as representing an exact decimal value in the usual “computerized scientific notation”.

Convert a Floating-Point Literal[26]:

```
temp = strnorm(c, 10, 10, "e");
CsmStrPtr = obstack_copy0(Csm_obstk, temp, strlen(temp));
```

This macro is invoked in definition 25.

Overflow is detected by subtracting the converted value from the maximum possible value; underflow is detected by subtracting the minimum possible value from the converted value. In both cases, a negative value represents the error condition. The constants for minimum and maximum values for float and double are taken from section 3.10.2 of the language specification.

Check for over- or underflow in a FloatLiteral[27]:

```
temp = strsub("3.40282347e38", CsmStrPtr, 10);
DomainError  = (!temp || temp[0] == '-');
if (CsmStrPtr[0] != '0') {
  /*
   * Fib on this one, since the JDK appears to have a smaller MIN_VALUE.
   *   temp = strsub(CsmStrPtr, "1.40239846e-45", 10);
   */
  temp = strsub(CsmStrPtr, "1.40129846432481707e-45", 10);
  DomainError |= (!temp || temp[0] == '-');
}
```

This macro is invoked in definition 25.

Check for over- or underflow in a DoubleLiteral[28]:

```
temp = strsub("1.79769313486231570e308", CsmStrPtr, 10);
DomainError  = (!temp || temp[0] == '-');
if (CsmStrPtr[0] != '0') {
  temp = strsub(CsmStrPtr, "4.94065645841246544e-324", 10);
  DomainError |= (!temp || temp[0] == '-');
}
```

This macro is invoked in definition 25.

1.10.3 Boolean Literals

The `boolean` type has two values, specified by keywords:

BooleanLiteral[29]:

```
'true' / 'false'
```

This macro is invoked in definition 39.

1.10.4 Character Literals

A literal of type `char` is expressed as a character or an escape sequence, enclosed in apostrophes:

Character Literal[30]:

```
CharacterLiteral: $'([^\\\n\015]|Escape[33]|Unicode Escape[3])' [mkchar]
```

This macro is invoked in definition 12.

Normalize a literal value[31]:

```
Token processor[1](‘mkchar’)
{ char strval[6], *p;
  int charval;

  if (c[1] != '\\') {
    strval[0] = c[1]; strval[1] = '\0';
    p = strval;
  } else if (c[2] == 'u') {
    p = strnorm(c + 3, 16, 10, "");
  } else if (c[2] >= '0' && c[2] <= '7') {
    p = strnorm(c + 2, 8, 10, "");
  } else {
    switch (c[2]) {
    case 'b': charval = '\b'; break;
    case 't': charval = '\t'; break;
    case 'n': charval = '\n'; break;
    case 'f': charval = '\f'; break;
    case 'r': charval = '\r'; break;
    case '"': charval = '"'; break;
    case '\'': charval = '\''; break;
    case '\\': charval = '\\'; break;
    default: {
      charval = c[2];
      message(ERROR, "Illegal escape", 0, &curpos);
      }
    }
    snprintf(strval, 6, "%d", charval);
    p = strval;
  }

  mkidn(p, strlen(p), t, s);
  *t = CharacterLiteral;
}
```

This macro is defined in definitions 18, 25, and 31.
This macro is invoked in definition 37.

1.10.5 String Literal

A string literal is zero or more characters enclosed in double quotes:

String Literal[32]:

```
StringLiteral:  $\"([^\\\n\r\"]|Escape[33]|Unicode Escape[3])*\"        [mkidn]
```

This macro is invoked in definition 12.

1.10.6 Escape Sequences for Character and String Literals

The character and string *escape sequences* allow for the representation of some nongraphic characters as well as the single quote, double quote, and backslash characters in character literals and string literals:

Escape[33]:

```
\\([btnfr\"'\\]|O[34]|O[34]O[34]|T[35]O[34]O[34])
```

This macro is invoked in definitions 30 and 32.

O[34]:

```
[0-7]
```

This macro is invoked in definition 33.

T[35]:

```
[0-3]
```

This macro is invoked in definition 33.

1.10.7 The Null Literal

The null type has one value, the null reference, denoted by a keyword:

NullLiteral[36]:

```
'null'
```

This macro is invoked in definition 39.

1.11 Separators

Separators do not have individual lexical descriptions in an Eli Specification. They appear as literal terminals in the grammar, and their lexical descriptions are extracted from the grammar by Eli.

1.12 Operators

Operators do not have individual lexical descriptions in an Eli Specification. They appear as literal terminals in the grammar, and their lexical descriptions are extracted from the grammar by Eli.

1.13 Support code

Scanner.c[37]:

```
#include "strmath.h"
#include "err.h"
#include "csm.h"
#include "termcode.h"

static void
```

```
#if PROTO_OK
NormInt(char *c, int l, int *t, int *s, int r)
#else
NormInt(c, l, t, s, r) char *c; int l, *t, *s, r;
#endif
```
*NormInt(char *c, int l, int *t, int *s, int r)*[19]

Report a keyword error[10]

Normalize a literal value[18]

This macro is attached to a product file.

Chapter 2

Phrase Structure

To solve the syntactic analysis subproblem, a Java processor must examine the sequence of tokens resulting from lexical analysis, grouping those tokens into a hierarchy of phrases. A context-free grammar is used to describe the phrase structure.

Character sequences that must appear exactly as specified are `'quoted'` in grammar rules. No additional lexical specification of these *literal terminals* is given. Non-literal terminals of the grammar are represented by symbols, and the form of the character strings acceptable as instances of these terminals is described by separate regular expressions.

The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by a colon. One or more alternative expansions for the nonterminal then follow, separated by slashes (`/`). The entire set of alternatives is terminated by a period.

Eli's notation for an optional symbol in a context-free grammar is to surround that symbol with square brackets (`[ ]`). In a regular expression, square brackets indicate a set of characters rather than an option. This corresponds to the "one of" notation found in the Java specification, but it cannot be used in context-free grammars.

A type-`con` file specifies the syntactic analysis subproblem:

Phrase.con[38]:

This macro is attached to a product file.

2.1 Productions from 3: Lexical Structure

Productions from 3: Lexical Structure[39]:

```
Literal:
  IntegerLiteral[17] /
  FloatingPointLiteral[24] /
  BooleanLiteral[29] /
  CharacterLiteral /
  StringLiteral /
  NullLiteral[36] .
```

This macro is invoked in definition 38.

2.2 Productions from 4: Types, Values, and Variables

Productions from 4: Types, Values, and Variables[40]:

```
Type:
  PrimitiveType /
  TypeName /
  ArrayType .

TypeName:
  Name .

PrimitiveType:
  NumericType /
  'boolean' .

NumericType:
  IntegralType /
  FloatingPointType .

IntegralType:
  'byte' / 'short' / 'int' / 'long' / 'char' .

FloatingPointType:
  'float' / 'double' .

ArrayType:
  PrimitiveType '[' ']' /
  Name '[' ']' /
  ArrayType '[' ']' .

ClassType:
  TypeName .

InterfaceType:
  InhName .
```

This macro is invoked in definition 38.

2.3 Productions from 6: Names

Productions from 6: Names[41]:

```
Name:
  Identifier /
  Name '.' Identifier .

PackageName: QualInhName .
InhName: QualInhName .

QualInhName:
  Identifier &'GotNameId(T_ATTR(TokenStack(0)));' /
  QualInhName '.' Identifier &'GotNameId(T_ATTR(TokenStack(2)));' .
```

This macro is invoked in definition 38.

2.4 Productions from 7: Packages

Productions from 7: Packages[42]:

```
CompilationUnit:
  PackageDeclarationOpt
  ImportJavaLang ImportDeclarationsOpt TypeDeclarationsOpt .

ImportJavaLang: .

ImportDeclarationsOpt: (&'StartName();' ImportDeclaration)* .

TypeDeclarationsOpt: TypeDeclaration* [';'] .

PackageDeclarationOpt:
  [&'StartName();' 'package' PackageName &'GotPackageImport();' ';'] .

ImportDeclaration:
  SingleTypeImportDeclaration ';' /
  TypeImportOnDemandDeclaration ';' .

SingleTypeImportDeclaration:
  'import' QualInhName &'GotTypeImport();' .

TypeImportOnDemandDeclaration:
  'import' QualInhName '.' '*' &'GotPackageImport();' .

TypeDeclaration:
  ClassDeclaration /
  InterfaceDeclaration .
```

This macro is invoked in definition 38.

2.5 Productions Only in the LALR(1) Grammar

Productions Only in the LALR(1) Grammar[43]:

```
Modifiers:
  / ModifierList .

ModifierList:
  Modifier / ModifierList Modifier .

Modifier:
  'public' / 'protected' / 'private' /
  'static' /
  'abstract' / 'final' / 'native' / 'synchronized' / 'transient' / 'volatile' /
  'strictfp' .
```

This macro is invoked in definition 38.

2.6 Productions from 8: Class Declarations

2.6.1 Productions from 8.1: Class Declaration

Productions from 8.1: Class Declaration[44]:

```
ClassDeclaration:
  Modifiers 'class' Identifier Super Interfaces ClassBody .

Super:
  'extends' InhName / .

Interfaces:
  ['implements' InterfaceTypeList] .

InterfaceTypeList:
  InterfaceType /
  InterfaceTypeList ',' InterfaceType .

ClassBody:
  '{' ClassBodyDeclarations '}' .

ClassBodyDeclarations:
  / ClassBodyDeclarationList .

ClassBodyDeclarationList:
  ClassBodyDeclaration /
  ClassBodyDeclarationList ClassBodyDeclaration .

ClassBodyDeclaration:
  ClassMemberDeclaration /
  ClassInitializer /
```

```
    ConstructorDeclaration .

  ClassMemberDeclaration:
    FieldDeclaration /
    MethodDeclaration /
    TypeDeclaration .
```

This macro is invoked in definition 38.

2.6.2 Productions from 8.3: Field Declarations

Declarations of fields and local variables have the same form, and the published grammar uses the same set of rules for both. The scope rules for fields are completely different from those for local variables, however, so it makes sense to distinguish their declarations syntactically.

Productions from 8.3: Field Declarations[45]:

```
  FieldDeclaration:
    Modifiers Type FieldDeclarators ';' .

  FieldDeclarators:
    FieldDeclarator /
    FieldDeclarators ',' FieldDeclarator .

  FieldDeclarator:
    FieldDeclaratorId /
    FieldDeclaratorId '=' Initializer .

  FieldDeclaratorId:
    FieldIdDef /
    FieldDeclaratorId '[' ']' .

  FieldIdDef:
    Identifier .

  Initializer:
    Expression /
    ArrayInitializer .
```

This macro is invoked in definition 38.

2.6.3 Productions from 8.4: Method Declarations

Productions from 8.4: Method Declarations[46]:

```
  MethodDeclaration:
    MethodHeader MethodBody .

  MethodHeader:
    Modifiers Type MethodDeclarator Throws /
    Modifiers Void MethodDeclarator Throws .
```

```
Void: 'void' .

MethodDeclarator:
  MethodIdDef '(' FormalParameters ')' /
  MethodDeclarator '[' ']' .

MethodIdDef:
  Identifier .

FormalParameters:
  [FormalParameterList] .

FormalParameterList:
  FormalParameter /
  FormalParameterList ',' FormalParameter .

FormalParameter:
  ['final'] Type VariableDeclaratorId .

Throws:
  / 'throws' ThrownTypeList .

ThrownTypeList:
  ThrownType /
  ThrownTypeList ',' ThrownType .

ThrownType: TypeName .

MethodBody:
  '{' StatementsOpt '}' /
  ';' .
```

This macro is invoked in definition 38.

2.6.4 Productions from 8.6 and 8.7: Class Initializers

Productions from 8.6 and 8.7: Class Initializers[47]:

```
ClassInitializer:
  ['static'] Block .
```

This macro is invoked in definition 38.

2.6.5 Productions from 8.8: Constructor Declarations

Productions from 8.8: Constructor Declarations[48]:

```
ConstructorDeclaration:
  Modifiers TypeName '(' FormalParameters ')' Throws
  '{' ConstructorStatements '}' .
```

```
ConstructorStatements:
  [ExplicitConstructorInvocation] [BlockStatements] .

ExplicitConstructorInvocation:
  'this' '(' Arguments ')' ';' /
  'super' '(' Arguments ')' ';' /
  Primary '.' 'super' '(' Arguments ')' ';' /
  Name '.' 'super' '(' Arguments ')' ';' .
```

This macro is invoked in definition 38.

2.7 Productions from 9: Interface Declarations

2.7.1 Productions from 9.1: Interface Declarations

Productions from 9.1: Interface Declarations[49]:

```
InterfaceDeclaration:
  Modifiers 'interface' Identifier ExtendsInterfaces InterfaceBody .

ExtendsInterfaces: ['extends' (InterfaceType // ',') ] .

InterfaceBody:
  '{' InterfaceMembers '}' .

InterfaceMembers:
  / InterfaceMemberDeclarations .

InterfaceMemberDeclarations:
  InterfaceMemberDeclaration /
  InterfaceMemberDeclarations InterfaceMemberDeclaration .

InterfaceMemberDeclaration:
  ConstantDeclaration /
  AbstractMethodDeclaration /
  TypeDeclaration .

ConstantDeclaration:
  FieldDeclaration .

AbstractMethodDeclaration:
  MethodHeader ';' .
```

This macro is invoked in definition 38.

2.8 Productions from 10: Arrays

Productions from 10: Arrays[50]:

```
ArrayInitializer:
  '{' Initializers '}' .

Initializers: [InitializerList] [','] .

InitializerList:
  Initializer /
  InitializerList ',' Initializer .
```

This macro is invoked in definition 38.

2.9 Productions from 14: Blocks and Statements

Productions from 14: Blocks and Statements[51]:

```
Block:
  '{' StatementsOpt '}' .

Statements:
  BlockStatements .

StatementsOpt:
  / BlockStatements .

BlockStatements:
  BlockStatement /
  BlockStatements BlockStatement .

BlockStatement:
  LocalVariableDeclarationStatement /
  Statement /
  TypeDeclaration .

LocalVariableDeclarationStatement:
  LocalVariableDeclaration ';' .

LocalVariableDeclaration:
  ['final'] Type VariableDeclarators .

VariableDeclarators:
  VariableDeclarator /
  VariableDeclarators ',' VariableDeclarator .

VariableDeclarator:
  VariableDeclaratorId /
  VariableDeclaratorId '=' Initializer .

VariableDeclaratorId:
  VariableIdDef /
```

```
  VariableDeclaratorId '[' ']' .

VariableIdDef:
  Identifier .

Statement:
  StatementWithoutTrailingSubstatement /
  LabeledStatement /
  IfThenStatement /
  IfThenElseStatement /
  WhileStatement /
  ForStatement .

StatementNoShortIf:
  StatementWithoutTrailingSubstatement /
  LabeledStatementNoShortIf /
  IfThenElseStatementNoShortIf /
  WhileStatementNoShortIf /
  ForStatementNoShortIf .

StatementWithoutTrailingSubstatement:
  Block /
  EmptyStatement /
  ExpressionStatement /
  SwitchStatement /
  DoStatement /
  BreakStatement /
  ContinueStatement /
  ReturnStatement /
  SynchronizedStatement /
  ThrowStatement /
  TryStatement /
  AssertStatement .

EmptyStatement:
    ';' .

LabeledStatement:
  Identifier ':' Statement .

LabeledStatementNoShortIf:
  Identifier ':' StatementNoShortIf .

ExpressionStatement:
  StatementExpression ';' .

StatementExpression:
  Assignment /
  PreIncrementExpression /
  PreDecrementExpression /
  PostIncrementExpression /
```

```
  PostDecrementExpression /
  MethodInvocation /
  ClassInstanceCreationExpression .

IfThenStatement:
  'if' '(' Expression ')' Statement .

IfThenElseStatement:
  'if' '(' Expression ')' StatementNoShortIf 'else' Statement .

IfThenElseStatementNoShortIf:
  'if' '(' Expression ')' StatementNoShortIf 'else' StatementNoShortIf .

SwitchStatement:
  'switch' '(' Expression ')' SwitchBlock .

SwitchBlock:
  '{' [SwitchBlockStatements] [SwitchLabels] '}' .

SwitchBlockStatements:
  SwitchBlockStatement /
  SwitchBlockStatements SwitchBlockStatement .

SwitchBlockStatement:
  SwitchLabels Statements .

SwitchLabels:
  SwitchLabel /
  SwitchLabels SwitchLabel .

SwitchLabel:
  'case' ConstantExpression ':' /
  'default' ':' .

WhileStatement:
  'while' '(' Expression ')' Statement .

WhileStatementNoShortIf:
  'while' '(' Expression ')' StatementNoShortIf .

DoStatement:
  'do' Statement 'while' '(' Expression ')' ';' .

ForStatement:
  'for' '(' ForInit ';' ForTest ';' ForUpdate ')'
    Statement .

ForStatementNoShortIf:
  'for' '(' ForInit ';' ForTest ';' ForUpdate ')'
    StatementNoShortIf .
```

```
ForInit:
  ExpressionList /
  LocalVariableDeclaration .

ForTest:
  / Expression .

ForUpdate:
  ExpressionList .

ExpressionList: / StatementExpressionList .

StatementExpressionList:
  StatementExpression /
  StatementExpressionList ',' StatementExpression .

BreakStatement:
  'break' [Identifier] ';' .

ContinueStatement:
  'continue' [Identifier] ';' .

ReturnStatement:
  'return' [Expression] ';' .

ThrowStatement:
  'throw' Expression ';' .

SynchronizedStatement:
  'synchronized' '(' Expression ')' Block .

TryStatement:
  'try' Block Catches .

Catches:
  CatchList /
  CatchesOpt Finally .

CatchesOpt:
  / CatchList .

CatchList:
  CatchClause /
  CatchList CatchClause .

CatchClause:
  'catch' '(' FormalParameter ')' Block .

Finally:
  'finally' Block .
```

```
AssertStatement:
  'assert' Expression ';' /
  'assert' Expression ':' Expression ';' .
```

This macro is invoked in definition 38.

2.10 Productions from 15: Expressions

Productions from 15: Expressions[52]:

```
Primary:
  PrimaryNoNewArray /
  ArrayCreationExpression .

PrimaryNoNewArray:
  Literal /
  Name '.' 'class' /
  'this' /
  Name '.' 'this' /
  '(' Expression ')' /
  ClassInstanceCreationExpression /
  FieldAccess /
  MethodInvocation /
  ArrayAccess .

ClassInstanceCreationExpression:
  Primary '.' 'new' Identifier '(' Arguments ')' AnonymousClass /
              'new' ClassType  '(' Arguments ')' AnonymousClass .

Arguments: [ Argument // ',' ] .

Argument: Expression .

AnonymousClass: ['{' ClassBodyDeclarations '}'] .

ArrayCreationExpression:
  'new' PrimitiveType Dimensions /
  'new' ClassType Dimensions .

Dimensions:
  DimExprList /
  DimExprList Dims .

DimExprList:
  DimExpr /
  DimExprList DimExpr .

DimExpr:
  '[' Expression ']' .
```

```
Dims:
  Dimension /
  Dims Dimension .

Dimension: '[' ']' .

FieldAccess:
  Primary '.' Identifier /
  'super' '.' Identifier /
  Name '.' 'super' '.' Identifier .

MethodInvocation:
  MethodName '(' Arguments ')' /
  Primary '.' Identifier '(' Arguments ')' /
  'super' '.' Identifier '(' Arguments ')' /
  Name '.' 'super' '.' Identifier '(' Arguments ')' .

MethodName:
  Name .

ArrayAccess:
  Name '[' Expression ']' /
  PrimaryNoNewArray '[' Expression ']' .

PostfixExpression:
  Primary /
  Name /
  PostIncrementExpression /
  PostDecrementExpression .

PostIncrementExpression:
  PostfixExpression '++' .

PostDecrementExpression:
  PostfixExpression '--' .

UnaryExpression:
  PreIncrementExpression /
  PreDecrementExpression /
  '+' UnaryExpression /
  '-' UnaryExpression /
  UnaryExpressionNotPlusMinus .

PreIncrementExpression:
  '++' UnaryExpression .

PreDecrementExpression:
  '--' UnaryExpression .

UnaryExpressionNotPlusMinus:
  PostfixExpression /
```

```
  '~' UnaryExpression /
  '!' UnaryExpression /
  CastExpression .

CastExpression:
  '(' PrimitiveType ')' UnaryExpression /
  '(' ArrayType ')'     UnaryExpressionNotPlusMinus /
  '(' Expression ')'    UnaryExpressionNotPlusMinus .

MultiplicativeExpression:
  UnaryExpression /
  MultiplicativeExpression '*' UnaryExpression /
  MultiplicativeExpression '/' UnaryExpression /
  MultiplicativeExpression '%' UnaryExpression .

AdditiveExpression:
  MultiplicativeExpression /
  AdditiveExpression '+' MultiplicativeExpression /
  AdditiveExpression '-' MultiplicativeExpression .

ShiftExpression:
  AdditiveExpression /
  ShiftExpression '<<' AdditiveExpression /
  ShiftExpression '>>' AdditiveExpression /
  ShiftExpression '>>>' AdditiveExpression .

RelationalExpression:
  ShiftExpression /
  RelationalExpression '<' ShiftExpression /
  RelationalExpression '>' ShiftExpression /
  RelationalExpression '<=' ShiftExpression /
  RelationalExpression '>=' ShiftExpression /
  RelationalExpression 'instanceof' Type .

EqualityExpression:
  RelationalExpression /
  EqualityExpression '==' RelationalExpression /
  EqualityExpression '!=' RelationalExpression .

AndExpression:
  EqualityExpression /
  AndExpression '&' EqualityExpression.

ExclusiveOrExpression:
  AndExpression /
  ExclusiveOrExpression '^' AndExpression.

InclusiveOrExpression:
  ExclusiveOrExpression /
  InclusiveOrExpression '|' ExclusiveOrExpression.
```

```
ConditionalAndExpression:
  InclusiveOrExpression /
  ConditionalAndExpression '&&' InclusiveOrExpression .

ConditionalOrExpression:
  ConditionalAndExpression /
  ConditionalOrExpression '||' ConditionalAndExpression .

ConditionalExpression:
  ConditionalOrExpression /
  ConditionalOrExpression '?' Expression ':' ConditionalExpression .

AssignmentExpression:
  ConditionalExpression /
  Assignment .

Assignment:
  LeftHandSide AssignmentOperator AssignmentExpression .

LeftHandSide:
  Name /
  FieldAccess /
  ArrayAccess .

AssignmentOperator:
  '=' / '*=' / '/=' / '%=' / '+=' / '-=' / '<<=' / '>>=' / '>>>=' / '&=' /
  '^=' / '|=' .

Expression:
  AssignmentExpression .

ConstantExpression:
  Expression .
```

This macro is invoked in definition 38.

2.11 Compiling multiple compilation units

A `Cluster` is a set of compilation units that is complete with respect to inheritance:

Compiling multiple compilation units[53]:

```
Goal:
  Cluster .

Cluster:
  ClusterElement / Cluster ClusterElement .

ClusterElement:
  (CompilationUnit 'EndOfFile')+ 'NoFilesPending' .
```

This macro is invoked in definition 38.

No compilation unit in the `Cluster` may inherit from a type available only in source form unless it is also a member of the `Cluster`. Unfortunately, the cluster is defined dynamically. If name analysis of the cluster reveals undefined types, the compiler must seek additional compilation units defining those types. In order to avoid this feedback from the name analysis, we obtain *all* files from packages mentioned in import declarations.

`NameStack` holds the components of a (qualified) name that might appear in a package or import declaration.

Instantiate necessary modules[54]:

```
$/Adt/Stack.gnrc +instance=Name +referto=int :inst
```

This macro is defined in definitions 54, 59, and 66.
This macro is invoked in definition 13.

`InName` is true when the parser is within an interesting name, and false otherwise:

State of import name collection[55]:

```
static int InName = 0;
```

This macro is invoked in definition 67.

Two routines collect each name:

void StartName(void)[56]:

```
/* Start collecting the identifiers of a (qualified) name
 *   On entry-
 *     The next identifier is the first in an interesting name
 *   On exit-
 *     NameStack is empty
 *     InName is true
 ***/
{ while (!NameStackEmpty) NameStackPop;
  InName = 1;
}
```

This macro is invoked in definition 67.

void GotNameId(int id)[57]:

```
/* Collect an identifier if necessary
 *   If InName on entry-
 *     NameStack contains the previous identifiers of the name
 *   If InName on exit-
 *     The name indexed by id has been added to NameStack
 ***/
{ if (InName) NameStackPush(id);
}
```

This macro is invoked in definition 67.

When the parser reaches the end of the name, it is handled in a manner dependent on the context:

GotPackageImport(void)[58]:

```
/* Handle a package import
 *   On entry-
 *     NameStack contains the complete sequence of identifiers for a name
 *   On exit-
 *     InName is false
 ***/
{ if (NameStackSize >= 1) {
    int idn = FQName();
    Binding bind = BindInScope(PkgRootEnv, idn);
    if (!HasTypScope(KeyOf(bind)))
      ImportPackage(StringTable(idn), KeyOf(bind));
  }

  InName = 0;
}
```

This macro is invoked in definition 67.

`PkgRootEnv` is defined as an environment with Algol scope rules, because each package is visible throughout the compilation:

Instantiate necessary modules[59]:

```
$/Name/AlgScope.gnrc +instance=Pkg +referto=Pkg :inst
```

This macro is defined in definitions 54, 59, and 66.
This macro is invoked in definition 13.

The `Package` property of a package is an environment with one binding for each type defined in the package. That binding's identifier is the simple name of the type, and its key is the key for the fully-qualified type.

Properties and access functions[60]:

```
TypScope: Environment [Has];    "envmod.h"
```

This macro is defined in definitions 60.
This macro is invoked in definition 65.

void GotTypeImport(void)[61]:

```
/* Handle a type import
 *   On entry-
 *     NameStack contains the complete sequence of identifiers for a name
 *   On exit-
 *     InName is false
 ***/
{ if (NameStackSize >= 2) {
    int idn = FQName();
    Binding bind = BindInScope(TypRootEnv, idn);
    ImportType(StringTable(idn));
```

```
  }

  InName = 0;
}
```

This macro is invoked in definition 67.

In both cases, the content of the name stack must be used to construct a fully-qualified name:

int FQName(void)[62]:

```
{ int i;

  for (i = 0; i < NameStackSize; i++) {
    char *s = StringTable(NameStackArray(i));

    if (i > 0) obstack_1grow(Csm_obstk,'.');
    obstack_grow(Csm_obstk, s, strlen(s));
  }
  obstack_1grow(Csm_obstk,'\0');
  CsmStrPtr = (char *)obstack_finish(Csm_obstk);

  return MakeName(CsmStrPtr);
}
```

This macro is invoked in definition 67.

The `EndOfFile` and `NoFilesPending` markers do not actually appear in the source text; they are deduced from conditions checked when the current input is exhausted. Therefore the corresponding literals must not be recognized as usual by the lexical analyzer:

Phrase.delit[63]:

```
$EndOfFile      EndOfFile
$NoFilesPending NoFilesPending
```

This macro is attached to a product file.

A token procesor that replaces the library end-of-file token processor recognizes these two markers:

*void EndOfText(char *c, int l, int *t, int *s)*[64]:

```
/* On entry-
 *   c points to the first character of the scanned string
 *   l=length of the scanned string
 *   *t=initial classification
 * On exit-
 *   *t=final classification
 *   *s=internal representation
 ***/

{ switch (EOTstate) {
  case 0:
    *t = EndOfFile; EOTstate = 1; return;
```

```
  case 1:
    if (AnotherCompilationUnit()) break;
    *t = NoFilesPending; EOTstate = 2; return;
  case 2:
    if (AnotherCompilationUnit()) break;
    return;
  }
  ResetScan = 1;
  *t = NORETURN; EOTstate = 0; return;
}
```

This macro is invoked in definition 67.

Phrase.pdl[65]:

```
IsDone: int;
```

Properties and access functions[60]

This macro is attached to a product file.

Instantiate necessary modules[66]:

```
$/Input/CoordMap.gnrc :inst
$/Tech/MakeName.gnrc +instance=Identifier :inst
```

This macro is defined in definitions 54, 59, and 66.
This macro is invoked in definition 13.

2.12 Support code

Parser.c[67]:

```
#include "csm.h"
#include "gla.h"
#include "litcode.h"
#include "pdl_gen.h"
#include "MakeName.h"
#include "NameStack.h"
#include "FilAlgScope.h"
#include "PkgAlgScope.h"
#include "TypAlgScope.h"
```

State of import name collection[55]

```
static int EOTstate = 0;

static int
#if PROTO_OK
FQName(void)
#else
```

```
FQName()
#endif
```
int FQName(void)[62]

```
void
#if PROTO_OK
StartName(void)
#else
StartName()
#endif
```
void StartName(void)[56]

```
void
#if PROTO_OK
GotNameId(int id)
#else
GotNameId(id) int id;
#endif
```
void GotNameId(int id)[57]

```
void
#if PROTO_OK
GotPackageImport(void)
#else
GotPackageImport()
#endif
```
GotPackageImport(void)[58]

```
void
#if PROTO_OK
GotTypeImport(void)
#else
GotTypeImport()
#endif
```
void GotTypeImport(void)[61]

```
void
#if PROTO_OK
EndOfText(char *c, int l, int *t, int *s)
#else
EndOfText(c, l, t, s) char *c; int l, *t, *s;
#endif
```
*void EndOfText(char *c, int l, int *t, int *s)*[64]

This macro is attached to a product file.

Chapter 3

Tree Structure

An attribute grammar is used to describe both the abstract syntax tree and the computations carried out over it. The fragment describing the overall structure of a Java AST serves to illustrate the notation:

Tree.lido[1]:

```
RULE: Goal    ::=    Cluster         END;
RULE: Cluster LISTOF CompilationUnit END;

RULE: CompilationUnit ::=
  PackageDeclarationOpt ImportJavaLang
  ImportDeclarationsOpt TypeDeclarationsOpt
END;

RULE: ImportJavaLang ::= END;

RULE pknm: PackageDeclarationOpt ::= 'package' PackageName ';' END;
RULE nonm: PackageDeclarationOpt ::=                          END;

RULE: ImportDeclarationsOpt LISTOF
  SingleTypeImportDeclaration | TypeImportOnDemandDeclaration
END;

RULE: TypeDeclarationsOpt LISTOF TypeDeclaration END;

RULE: TypeImportOnDemandDeclaration ::= 'import' QualInhName '.' '*' END;
RULE: SingleTypeImportDeclaration   ::= 'import' QualInhName         END;

RULE: QualInhName ::=                 InhBaseId END;
RULE: QualInhName ::= QualInhName '.' InhQualId END;

RULE: InhBaseId ::= Identifier END;
RULE: InhQualId ::= Identifier END;
```

Abstract syntax[2]

This macro is attached to a product file.

Each rule describes a node of the AST, and corresponds to a class in an object-oriented implementation. The identifier following the keyword `RULE` names the class of the object that would represent such a node. These are the only classes that can be instantiated. (It is not necessary to name the rules in a LIDO specification, because Eli will generate unique names if none are given, but rules can be named to make it easier to discuss them.)

An identifier that precedes `::=` or `LISTOF` in some rule is called a *nonterminal*; all other identifiers are *terminals*. Nonterminals following `::=` represent subtrees, while terminals represent values that are not subtrees. (For example, `Identifier` is a terminal representing an identifier appearing in the source program.)

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

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

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

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

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

A `CompilationUnit` constitutes the text in a single file. Types declared in different compilation units can depend on each other, circularly. The compiler must compile such types all at the same time, which implies that it must deal with an arbitrary number of compilation units:

3.1 Abstract syntax

Abstract syntax[2]:

```
RULE: TypeDeclaration ::=
        Modifiers 'class'     TypeIdDef Super Interfaces ClassBody     END;
RULE: TypeDeclaration ::=
        Modifiers 'interface' TypeIdDef       Interfaces InterfaceBody END;

RULE: Modifiers LISTOF Modifier       END;
RULE: Modifier  ::=    'abstract'     END;
RULE: Modifier  ::=    'final'        END;
RULE: Modifier  ::=    'native'       END;
RULE: Modifier  ::=    'private'      END;
RULE: Modifier  ::=    'protected'    END;
RULE: Modifier  ::=    'public'       END;
RULE: Modifier  ::=    'static'       END;
RULE: Modifier  ::=    'synchronized' END;
RULE: Modifier  ::=    'transient'    END;
RULE: Modifier  ::=    'volatile'     END;
RULE: Modifier  ::=    'strictfp'     END;

RULE: Super ::= 'extends' InhName END;
```

```
RULE: Super ::=                    END;

RULE: Interfaces LISTOF InterfaceType END;

RULE: InterfaceType ::= InhName     END;
RULE: InhName       ::= QualInhName END;

RULE: ClassBody             ::=    '{' ClassBodyDeclarations '}' END;
RULE: ClassBodyDeclarations LISTOF FieldDeclaration |
                                   MethodDeclaration |
                                   TypeDeclaration |
                                   ConstructorDeclaration |
                                   ClassInitializer            END;

RULE: ConstructorDeclaration ::=
        Modifiers TypeName '(' FormalParameters ')' Throws
        '{' Statements '}' END;

RULE: InterfaceBody ::= '{' InterfaceMembers '}' END;
RULE: InterfaceMembers LISTOF AbstractMethodDeclaration |
                              TypeDeclaration |
                              ConstantDeclaration END;

RULE: AbstractMethodDeclaration ::= MethodHeader ';' END;
RULE: ConstantDeclaration       ::= FieldDeclaration END;

RULE: Statements LISTOF LocalVariableDeclaration |
                        Statement |
                        TypeDeclaration END;

RULE: FieldDeclaration  ::=    Modifiers Type FieldDeclarators ';' END;
RULE: FieldDeclarators  LISTOF FieldDeclarator                     END;
RULE: FieldDeclarator   ::=    FieldDeclaratorId                   END;
RULE: FieldDeclarator   ::=    FieldDeclaratorId '=' Initializer   END;
RULE: FieldDeclaratorId ::=    FieldIdDef                          END;

RULE: MethodDeclaration ::=    MethodHeader MethodBody                END;
RULE: MethodHeader      ::=    Modifiers Type MethodDeclarator Throws END;
RULE: MethodDeclarator  ::=    MethodIdDef '(' FormalParameters ')'   END;
RULE: MethodDeclarator  ::=    MethodDeclarator '[' ']'               END;
RULE: FormalParameters  LISTOF FormalParameter                        END;
RULE: FormalParameter   ::=   'final' Type VariableDeclaratorId       END;
RULE: FormalParameter   ::=           Type VariableDeclaratorId       END;
RULE: MethodBody        ::=    '{' Statements '}'                     END;
RULE: MethodBody        ::=    ';'                                    END;

RULE: LocalVariableDeclaration ::= 'final' Type VariableDeclarators END;
RULE: LocalVariableDeclaration ::=         Type VariableDeclarators END;

RULE: Expression ::= CharacterLiteral END;
RULE: Expression ::= DoubleLiteral END;
```

```
RULE: Expression ::= '(' PrimitiveType ')' Expression END;
RULE: Expression ::= '(' ArrayType ')'     Expression END;
RULE: Expression ::=            Operator Expression END;
RULE: Expression ::= Expression Operator            END;
RULE: Expression ::= Expression Operator Expression END;
RULE: Expression ::= Expression '?' Expression ':' Expression END;
RULE: Expression ::= Expression 'instanceof' Type END;
RULE: Expression ::= Expression '.' MethodIdUse '(' Arguments ')' END;
RULE: Expression ::= Expression '.' FieldIdUse END;
RULE: Expression ::= 'false' END;
RULE: Expression ::= FloatLiteral END;
RULE: Expression ::= IntLiteral END;
RULE: Expression ::= LeftHandSide AssignmentOperator RightHandSide END;
RULE: Expression ::= LongLiteral END;
RULE: Expression ::= MethodName '(' Arguments ')' END;
RULE: Expression ::=
  Expression '.' 'new' TypeIdUse '(' Arguments ')' AnonymousClass END;
RULE: Expression ::= 'new' TypeName '(' Arguments ')' AnonymousClass END;
RULE: Expression ::= 'new' PrimitiveType Dimensions END;
RULE: Expression ::= 'new' TypeName      Dimensions END;
RULE: Expression ::= 'null' END;
RULE: Expression ::= StringLiteral END;
RULE: Expression ::= 'super' '.' MethodIdUse '(' Arguments ')' END;
RULE: Expression ::= 'super' '.' FieldIdUse END;
RULE: Expression ::=              'this' END;
RULE: Expression ::= 'true' END;
RULE: LeftHandSide ::= Expression END;
RULE: RightHandSide ::= Expression END;
RULE: Arguments LISTOF Argument END;
RULE: Argument ::= Expression END;

RULE: AnonymousClass ::= '{' ClassBodyDeclarations '}' END;
RULE: AnonymousClass ::=                               END;
RULE: Dimensions LISTOF Dimension END;
RULE: Dimension ::= '[' Expression ']' END;
RULE: Dimension ::= '[' ']' END;

RULE: ExpressionStatement ::= Expression               ';' END;
RULE: ExpressionStatement ::= 'this'  '(' Arguments ')' ';' END;
RULE: ExpressionStatement ::= Expression '.' 'super' '(' Arguments ')' ';' END;
RULE: ExpressionStatement ::=               'super' '(' Arguments ')' ';' END;

RULE: Block ::= '{' Statements '}' END;

RULE: LabeledStatement ::= LabelIdDef ':' Statement END;

RULE: Statement ::= AssertStatement END;
RULE: Statement ::= Block END;
RULE: Statement ::= ExpressionStatement END;
RULE: Statement ::= 'break' ';' END;
RULE: Statement ::= 'break' LabelIdUse ';' END;
```

```
RULE: Statement ::= 'continue' ';' END;
RULE: Statement ::= 'continue' LabelIdUse ';' END;
RULE: Statement ::= LoopStatement END;
RULE: Statement ::= ';' END;
RULE: Statement ::= 'if' '(' Expression ')' Statement 'else' Statement END;
RULE: Statement ::= 'if' '(' Expression ')' Statement END;
RULE: Statement ::= LabeledStatement END;
RULE: Statement ::= 'return' ';' END;
RULE: Statement ::= 'return' Expression ';' END;
RULE: Statement ::= SwitchStatement END;
RULE: Statement ::= 'synchronized' '(' Expression ')' Block END;
RULE: Statement ::= 'throw' Expression ';' END;
RULE: Statement ::= 'try' Block Catches END;

RULE: LoopStatement ::= WhileStatement END;
RULE: LoopStatement ::= DoStatement END;
RULE: LoopStatement ::= ForStatement END;

RULE: AssertStatement ::= 'assert' Expression ';' END;
RULE: AssertStatement ::= 'assert' Expression ':' Expression ';' END;

RULE: DoStatement ::= 'do' Statement 'while' '(' Expression ')' ';' END;

RULE: ForStatement ::= 'for' '(' ForInit ';' ForTest ';' ForUpdate ')' Statement END;
RULE: ForInit ::= LocalVariableDeclaration END;
RULE: ForInit ::= ExpressionList END;
RULE: ForTest ::= Expression END;
RULE: ForTest ::= END;
RULE: ForUpdate ::= ExpressionList END;

RULE: ExpressionList LISTOF Expression END;

RULE: SwitchStatement ::= 'switch' '(' Expression ')' SwitchBlock END;
RULE: SwitchBlock ::= '{' '}' END;
RULE: SwitchBlock ::= '{' SwitchBlockStatements '}' END;
RULE: SwitchBlock ::= '{' SwitchBlockStatements SwitchLabels '}' END;
RULE: SwitchBlock ::= '{' SwitchLabels '}' END;
RULE: SwitchBlockStatements ::= SwitchBlockStatement END;
RULE: SwitchBlockStatements ::= SwitchBlockStatements SwitchBlockStatement END;
RULE: SwitchBlockStatement ::= SwitchLabels Statements END;
RULE: SwitchLabels LISTOF SwitchLabel END;
RULE: SwitchLabel ::= 'case' Expression ':' END;
RULE: SwitchLabel ::= 'default' ':' END;

RULE: WhileStatement ::= 'while' '(' Expression ')' Statement END;

RULE: Catches LISTOF CatchClause | Finally END;
RULE: CatchClause ::= 'catch' '(' FormalParameter ')' Block END;
RULE: Finally ::= 'finally' Block END;

RULE: ClassInitializer ::= 'static' Block END;
```

```
RULE: ClassInitializer ::=          Block END;

RULE: Throws LISTOF ThrownType END;
RULE: ThrownType ::= TypeName END;

RULE: PrimitiveType ::= 'boolean' END;
RULE: PrimitiveType ::= 'byte'    END;
RULE: PrimitiveType ::= 'char'    END;
RULE: PrimitiveType ::= 'double'  END;
RULE: PrimitiveType ::= 'float'   END;
RULE: PrimitiveType ::= 'int'     END;
RULE: PrimitiveType ::= 'long'    END;
RULE: PrimitiveType ::= 'short'   END;

RULE: ArrayType ::= PrimitiveType '[' ']' END;
RULE: ArrayType ::= ArrayType     '[' ']' END;

RULE: Type ::= PrimitiveType END;
RULE: Type ::= TypeName      END;
RULE: Type ::= ArrayType     END;
RULE: Type ::= 'void'        END;

RULE: VariableDeclarators LISTOF VariableDeclarator END;

RULE: VariableDeclarator ::= VariableDeclaratorId END;
RULE: VariableDeclarator ::= VariableDeclaratorId '=' Initializer END;

RULE: VariableDeclaratorId ::= VariableDeclaratorId '[' ']' END;
RULE: VariableDeclaratorId ::= VariableIdDef                END;

RULE: Initializer ::= Expression       END;
RULE: Initializer ::= '{' Initializers '}' END;

RULE: Initializers LISTOF InitialElement END;
RULE: InitialElement ::= Initializer END;

RULE: AssignmentOperator ::= '^='   END;
RULE: AssignmentOperator ::= '<<='  END;
RULE: AssignmentOperator ::= '='    END;
RULE: AssignmentOperator ::= '>>='  END;
RULE: AssignmentOperator ::= '>>>=' END;
RULE: AssignmentOperator ::= '|='   END;
RULE: AssignmentOperator ::= '-='   END;
RULE: AssignmentOperator ::= '/='   END;
RULE: AssignmentOperator ::= '*='   END;
RULE: AssignmentOperator ::= '&='   END;
RULE: AssignmentOperator ::= '%='   END;
RULE: AssignmentOperator ::= '+='   END;
RULE: Operator ::= '^' END;
RULE: Operator ::= '<<' END;
RULE: Operator ::= '<=' END;
```

```
RULE: Operator ::= '<' END;
RULE: Operator ::= '==' END;
RULE: Operator ::= '>=' END;
RULE: Operator ::= '>>>' END;
RULE: Operator ::= '>>' END;
RULE: Operator ::= '>' END;
RULE: Operator ::= '||' END;
RULE: Operator ::= '|' END;
RULE: Operator ::= '+' END;
RULE: Operator ::= '-' END;
RULE: Operator ::= '!=' END;
RULE: Operator ::= '/' END;
RULE: Operator ::= '*' END;
RULE: Operator ::= '&' END;
RULE: Operator ::= '&&' END;
RULE: Operator ::= '%' END;
RULE: Operator ::= '++' END;
RULE: Operator ::= '--' END;
RULE: Operator ::= '~' END;
RULE: Operator ::= '!' END;

RULE: VariableIdDef ::= Identifier END;
RULE: MethodIdDef ::= Identifier END;
RULE: MethodIdUse ::= Identifier END;
RULE: LabelIdDef ::= Identifier END;
RULE: LabelIdUse ::= Identifier END;
RULE: FieldIdDef ::= Identifier END;
RULE: FieldIdUse ::= Identifier END;
RULE: TypeIdDef ::= Identifier END;
RULE: TypeIdUse ::= Identifier END;
```

Disambiguating names[3]

This macro is invoked in definition 1.

3.2 Disambiguating names

Disambiguating names[3]:

```
RULE: Name ::=          Identifier END;
RULE: Name ::= Name '.' Identifier END;
```

Type names[4]
Expression names[8]
Method names[11]
Ambiguous names[14]

This macro is defined in definitions 3 and 17.
This macro is invoked in definition 2.

3.2.1 Type names

Type names[4]:

```
RULE: TypeName ::= Name $pTypeName COMPUTE
  pTypeName.GENTREE=TP_tpTypeName(Name.tp);
END;

RULE: Expression ::= Name $pTypeName '.' 'class' COMPUTE
  pTypeName.GENTREE=TP_tpTypeName(Name.tp);
END;

RULE: Expression ::= Name $pTypeName '.' 'super'
        '.' MethodIdUse '(' Arguments ')'
COMPUTE
  pTypeName.GENTREE=TP_tpTypeName(Name.tp);
END;

RULE: Expression ::= Name $pTypeName '.' 'super'
        '.' FieldIdUse
COMPUTE
  pTypeName.GENTREE=TP_tpTypeName(Name.tp);
END;

RULE: Expression ::= Name $pTypeName '.' 'this' COMPUTE
  pTypeName.GENTREE=TP_tpTypeName(Name.tp);
END;

RULE: Expression ::= '(' Expression $pTypeName ')' Expression COMPUTE
  pTypeName.GENTREE=TP_tpTypeName(Expression[2].tp);
  Expression[2].IsCastType=1;
END;

RULE stypnm: pTypeName ::=               sTypeIdUse END;
RULE qtypnm: pTypeName ::= pPkgOrTypName qTypeIdUse END;

RULE spotnm: pPkgOrTypName ::= sTypeIdUse END;
RULE qpotnm: pPkgOrTypName ::= pPkgOrTypName qTypeIdUse END;

RULE stypid: sTypeIdUse ::= Identifier END;
RULE qtypid: qTypeIdUse ::= Identifier END;
```

This macro is defined in definitions 4 and 7.
This macro is invoked in definition 3.

Tree parsing rules[5]:

```
tpTypeName ::= tpId(int): mkstypnm;
tpTypeName ::= tpDot(tpPkgOrTypName,int): mkqtypnm;

tpPkgOrTypName ::= tpId(int): mkspotnm;
tpPkgOrTypName ::= tpDot(tpPkgOrTypName,int): mkqpotnm;
```

This macro is defined in definitions 5, 9, 12, 15, and 19.
This macro is invoked in definition 18.

Tree parsing actions[6]:

```
#define mkstypnm(x)\
  Mkstypnm(NoPosition,Mkstypid(NoPosition,x))
#define mkqtypnm(x,y)\
  Mkqtypnm(NoPosition,x,Mkqtypid(NoPosition,y))

#define mkspotnm(x)\
  Mkspotnm(NoPosition,Mkstypid(NoPosition,x))
#define mkqpotnm(x,y)\
  Mkqpotnm(NoPosition,x,Mkqtypid(NoPosition,y))
```

This macro is defined in definitions 6, 10, 13, and 16.
This macro is invoked in definition 21.

Type names[7]:

```
RULE: ArrayType ::= Name $pTypeName '[' ']' COMPUTE
  pTypeName.GENTREE=TP_tpTypeName(Name.tp);
END;
```

This macro is defined in definitions 4 and 7.
This macro is invoked in definition 3.

3.2.2 Expression names

Expression names[8]:

```
RULE: ExpressionStatement ::= Name $pExpressionName
        '.' 'super' '(' Arguments ')' ';' COMPUTE
  pExpressionName.GENTREE=TP_tpExpressionName(Name.tp);
END;

RULE: Expression ::= Name $pExpressionName '[' Expression ']' COMPUTE
  pExpressionName.GENTREE=TP_tpExpressionName(Name.tp);
END;

TREE SYMBOL Expression:
  IsCastType: int,
  tp: TPNode;

TREE SYMBOL Expression COMPUTE
  SYNT.tp=TPNULL;
  INH.IsCastType=0;
END;

RULE: Expression ::= Name $pExpressionName COMPUTE
  Expression.tp=IF(Expression.IsCastType,Name.tp,TPNULL);
  pExpressionName.GENTREE=
    IF(Expression.IsCastType,
      Mknexpnm(COORDREF),
      TP_tpExpressionName(Name.tp));
END;

RULE: LeftHandSide ::= Name $pExpressionName COMPUTE
```

```
  pExpressionName.GENTREE=TP_tpExpressionName(Name.tp);
END;

RULE nexpnm: pExpressionName ::=                          END;
RULE sexpnm: pExpressionName ::=                sExprIdUse END;
RULE qexpnm: pExpressionName ::= pAmbiguousName qExprIdUse END;

RULE sexpid: sExprIdUse ::= Identifier END;
RULE qexpid: qExprIdUse ::= Identifier END;
```

This macro is invoked in definition 3.

Tree parsing rules[9]:

```
tpExpressionName ::= tpId(int): mksexpnm;
tpExpressionName ::= tpDot(tpAmbiguousName,int): mkqexpnm;
```

This macro is defined in definitions 5, 9, 12, 15, and 19.
This macro is invoked in definition 18.

Tree parsing actions[10]:

```
#define mksexpnm(x)\
  Mksexpnm(NoPosition,Mksexpid(NoPosition,x))
#define mkqexpnm(x,y)\
  Mkqexpnm(NoPosition,x,Mkqexpid(NoPosition,y))
```

This macro is defined in definitions 6, 10, 13, and 16.
This macro is invoked in definition 21.

3.2.3 Method names

Method names[11]:

```
RULE: MethodName ::= Name $pMethodName COMPUTE
  pMethodName.GENTREE=TP_tpMethodName(Name.tp);
END;

RULE smthnm: pMethodName ::= sMethIdUse END;
RULE qmthnm: pMethodName ::= pAmbiguousName qMethIdUse END;

RULE smthid: sMethIdUse ::= Identifier END;
RULE qmthid: qMethIdUse ::= Identifier END;
```

This macro is invoked in definition 3.

Tree parsing rules[12]:

```
tpMethodName ::= tpId(int): mksmthnm;
tpMethodName ::= tpDot(tpAmbiguousName,int): mkqmthnm;
```

This macro is defined in definitions 5, 9, 12, 15, and 19.
This macro is invoked in definition 18.

Tree parsing actions[13]:

```
#define mksmthnm(x)\
  Mksmthnm(NoPosition,Mksmthid(NoPosition,x))
#define mkqmthnm(x,y)\
  Mkqmthnm(NoPosition,x,Mkqmthid(NoPosition,y))
```

This macro is defined in definitions 6, 10, 13, and 16.
This macro is invoked in definition 21.

3.2.4 Ambiguous names

Ambiguous names[14]:

```
RULE sambnm: pAmbiguousName ::= sAmbgIdUse END;
RULE qambnm: pAmbiguousName ::= pAmbiguousName qAmbgIdUse END;

RULE sambid: sAmbgIdUse ::= Identifier END;
RULE qambid: qAmbgIdUse ::= Identifier END;
```

This macro is invoked in definition 3.

Tree parsing rules[15]:

```
tpAmbiguousName ::= tpId(int): mksambnm;
tpAmbiguousName ::= tpDot(tpAmbiguousName,int): mkqambnm;
```

This macro is defined in definitions 5, 9, 12, 15, and 19.
This macro is invoked in definition 18.

Tree parsing actions[16]:

```
#define mksambnm(x)\
  Mksambnm(NoPosition,Mksambid(NoPosition,x))
#define mkqambnm(x,y)\
  Mkqambnm(NoPosition,x,Mkqambid(NoPosition,y))
```

This macro is defined in definitions 6, 10, 13, and 16.
This macro is invoked in definition 21.

Disambiguating names[17]:

```
ATTR tp: TPNode;

RULE: Name ::= Identifier COMPUTE
  Name.tp=TP_0_int(tpId,Identifier);
END;

RULE: Name ::= Name '.' Identifier COMPUTE
  Name[1].tp=TP_1_int(tpDot,Name[2].tp,Identifier);
END;
```

This macro is defined in definitions 3 and 17.
This macro is invoked in definition 2.

Tree.tp[18]:

Tree parsing rules[5]

```
tpAmbiguousName ::= tpNever(tpAmbiguousName,tpAmbiguousName): nop;
```

This macro is attached to a product file.

Tree parsing rules[19]:

```
tpTypeName, tpExpressionName, tpMethodName, tpPkgOrTypName,
tpAmbiguousName: NODEPTR;       "treecon.h"
```

This macro is defined in definitions 5, 9, 12, 15, and 19.
This macro is invoked in definition 18.

Tree.head[20]:

```
#include "Tree.h"
```

This macro is attached to a product file.

Tree.h[21]:

```
#ifndef TREE_H
#define TREE_H

#include "treecon.h"

#define nop(x,y)
```

Tree parsing actions[6]

```
#endif
```

This macro is attached to a product file.

3.3 Relating Phrase Structure to Tree Structure

Although the phrase structure specified by the context-free grammar and the tree structure implicit in the attribute grammar are related, they are not identical. Eli determines their relationship by comparing the structures of the specifications and taking account of additional information provided by the user in a special mapping language.

The phrase structure of Java is completely specified by the context-free grammar contained in `Phrase.con` (defined above). A user can obtain this grammar from Eli by requesting the `consyntax` product. `Tree.lido` (defined above), on the other hand, specifies only those contexts in which computations take place. Eli deduces the remainder of the tree structure from `Phrase.con` and `Tree.map` (defined below). A user can obtain the complete set of rules describing the tree structure from Eli by requesting the `abstree` product.

There are two kinds of descriptions in the mapping language: symbol mappings and rule mappings. Symbol mappings specify that a number of symbols representing distinct phrases correspond to a single symbol representing a distinct tree node. Only the tree node symbol may appear in the attribute grammar. Rule mappings specify that a given phrase corresponds to a particular tree fragment. Only the tree fragments may appear in the attribute grammar.

A type-`map` file describes correspondences between the phrase structure of the input text and the tree that represents the program internally.

Tree.map[22]:

```
MAPSYM
```
Types[23]
Statements[25]
Expressions[24]

This macro is attached to a product file.

3.3.1 Types

Some type errors can be detected when the program is parsed, by using distinct grammar symbols to represent different subsets of the type universe. Once parsing is complete, however, there is no need to retain these distinctions as different kinds of tree node. Every type name is therefore represented in the tree by a `Type` node that has a `TypeKey` attribute to hold the definition table key representing that type internally:

Types[23]:

```
PrimitiveType ::= NumericType IntegralType FloatingPointType .

Type ::= Void .

TypeName ::= ClassType .

TypeDeclaration ::= ClassDeclaration InterfaceDeclaration .

Interfaces ::= ExtendsInterfaces .
```

This macro is invoked in definition 22.

3.3.2 Expressions

Precedence and association are expressed in the phrase structure by distinct nonterminals. These concepts have to do with the relationship between the linear text and the semantic structure, and are no longer relevant once the AST has been built. Retaining distinctions between semantically-equivalent nonterminals in the AST is counterproductive:

Expressions[24]:

```
Expression ::=
  Primary PrimaryNoNewArray ArrayCreationExpression Literal
  ClassInstanceCreationExpression FieldAccess MethodInvocation ArrayAccess
  PostfixExpression PostIncrementExpression PostDecrementExpression
  UnaryExpression PreIncrementExpression PreDecrementExpression
  UnaryExpressionNotPlusMinus CastExpression MultiplicativeExpression
  AdditiveExpression ShiftExpression RelationalExpression EqualityExpression
  AndExpression ExclusiveOrExpression InclusiveOrExpression
  ConditionalAndExpression ConditionalOrExpression ConditionalExpression
  AssignmentExpression Assignment StatementExpression ConstantExpression .

Initializer ::= ArrayInitializer .
Dimension  ::= DimExpr .
```

This macro is invoked in definition 22.

3.3.3 Statements

The concrete syntax for Java introduces a large number of nonterminals for the purpose of removing the "dangling else" ambiguity, and also to differentiate statements for the purposes of exposition. In most of these cases no special semantics are associated with these nonterminals, and therefore there is no need to distinguish them in the tree:

Statements[25]:

```
Statements ::= StatementsOpt ConstructorStatements .

Statement ::=
  StatementWithoutTrailingSubstatement StatementNoShortIf
  IfThenStatement IfThenElseStatement IfThenElseStatementNoShortIf
  BreakStatement ContinueStatement
  EmptyStatement
  ReturnStatement
  SynchronizedStatement
  ThrowStatement
  TryStatement .

ExpressionStatement ::=
  ExplicitConstructorInvocation .

LabeledStatement ::= LabeledStatementNoShortIf .
WhileStatement   ::= WhileStatementNoShortIf .
ForStatement     ::= ForStatementNoShortIf .
```

This macro is invoked in definition 22.

Chapter 4

Name Analysis

Names are used to refer to entities declared in a Java program. A declared entity is a package, class type, interface type, member (field or method) of a reference type, parameter (to a method, constructor, or exception handler), or local variable.

Every name introduced by a declaration has a *scope*, which is the part of the Java program text within which the declared entity can be referred to by a simple name.

This specification implements the scope rules of Java. It uses a number of Eli modules specifying computational roles that are useful in name analysis.

Name.specs[1]:

> *Instantiate required modules*[3]

This macro is attached to a product file.

A precondition for using any of Eli's name analysis modules is that every identifier occurrence have a `Sym` attribute:

Name.lido[2]:

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

Packages[4]
Scopes[11]
Applied occurrences of names[21]
Labeled Statements[35]

This macro is attached to a product file.

The same identifier can be used in a specific context to denote a package, a type, a method, or a field. Although a local variable name can hide a field name, local variables and fields obey different scope rules. Thus each of the five kinds of entities has its own name space, implemented by an instantiation of an Eli name analysis module. All of the computational roles of a specific module instance are qualified by the instance name.

Instantiate required modules[3]:

```
$/Name/AlgScope.gnrc +instance=Typ +referto=Typ :inst
$/Name/AlgInh.gnrc   +instance=Typ +referto=Typ :inst
```

```
$/Name/ScopeProp.gnrc+instance=Typ +referto=Typ :inst

$/Name/AlgScope.gnrc +instance=Fld +referto=Fld :inst
$/Name/AlgInh.gnrc   +instance=Fld +referto=Fld :inst
$/Name/ScopeProp.gnrc+instance=Fld +referto=Fld :inst

$/Name/AlgScope.gnrc +instance=Mth +referto=Mth :inst
$/Name/AlgInh.gnrc   +instance=Mth +referto=Mth :inst
$/Name/ScopeProp.gnrc+instance=Mth +referto=Mth :inst

$/Name/CScope.gnrc   +instance=Var +referto=Var :inst
```

This macro is defined in definitions 3, 6, 7, 27, 30, and 36.
This macro is invoked in definition 1.

4.1 Packages

A package consists of a number of compilation units. A compilation unit automatically has access to all types declared in its package. To model this behavior, we associate a definition table key with each package, and make that key available in each compilation unit via the compilation unit's `PackageDeclarationOpt` child:

Packages[4]:

```
TREE SYMBOL PackageDeclarationOpt: PkgKey: DefTableKey;

TREE SYMBOL PackageName INHERITS PkgIdDefScope END;

RULE: PackageDeclarationOpt ::= 'package' PackageName ';' COMPUTE
  PackageDeclarationOpt.PkgKey=PackageName.PkgKey;
END;
```

This macro is defined in definitions 4, 5, 8, and 9.
This macro is invoked in definition 2.

A compilation unit that has no package declaration is part of an unnamed package. We have chosen to support an arbitrary number of unnamed packages, each containing a single compilation unit:

Packages[5]:

```
RULE: PackageDeclarationOpt ::= COMPUTE
  PackageDeclarationOpt.PkgKey=NewKey();
END;
```

This macro is defined in definitions 4, 5, 8, and 9.
This macro is invoked in definition 2.

Instantiate required modules[6]:

```
$/Name/AlgScope.gnrc +instance=Pkg +referto=Pkg :inst
```

This macro is defined in definitions 3, 6, 7, 27, 30, and 36.
This macro is invoked in definition 1.

The members of a package are class and interface types, which are declared in compilation units of the package. We model this behavior by the `Pty` name space.

Instantiate required modules[7]:

```
$/Name/AlgScope.gnrc +instance=Pty +referto=Pty :inst
```

This macro is defined in definitions 3, 6, 7, 27, 30, and 36.
This macro is invoked in definition 1.

All of the declarations of package members are children of the `TypeDeclarationsOpt` child of a compilation unit. The `Pty` name space must, however, define the full set of type scopes to hide lower-level definitions:

Packages[8]:

```
TREE SYMBOL TypeDeclarationsOpt  INHERITS PtyRangeScope END;
TREE SYMBOL ClassBody            INHERITS PtyRangeScope END;
TREE SYMBOL InterfaceBody        INHERITS PtyRangeScope END;
TREE SYMBOL MethodDeclaration    INHERITS PtyRangeScope END;
TREE SYMBOL Block                INHERITS PtyRangeScope END;
TREE SYMBOL SwitchBlockStatement INHERITS PtyRangeScope END;

TREE SYMBOL TypeIdDef            INHERITS PtyIdDefScope END;
```

This macro is defined in definitions 4, 5, 8, and 9.
This macro is invoked in definition 2.

A single environment is used for all compilation units in a single package. The package key is used to retrieve that environment:

Packages[9]:

```
RULE: CompilationUnit ::=
        PackageDeclarationOpt
        ImportJavaLang ImportDeclarationsOpt TypeDeclarationsOpt
COMPUTE
  TypeDeclarationsOpt.PtyEnv=
    PackagePtyScope(PackageDeclarationOpt.PkgKey,INCLUDING Goal.PtyEnv);
END;
```

This macro is defined in definitions 4, 5, 8, and 9.
This macro is invoked in definition 2.

`PackagePtyScope` obtains the package's environment from the definition table, if it exists. Otherwise, it returns a new environment and stores that environment as the package's environment:

Name.pdl[10]:

```
PtyScope: Environment [Package];        "envmod.h"

Environment Package(DefTableKey key, Environment env)
{ if (key == NoKey) return NoEnv;
  if (!ACCESS) VALUE = NewScope(env);
  return VALUE;
}
```

This macro is attached to a product file.

4.2 Scope of a simple name

The *scope* of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name.

4.2.1 Imported types

FIXME: This is actually an incorrect rendition of the import policy. A `SingleTypeImportDeclaration` should act precisely like the declaration of a class or interface in this compilation unit. In order to implement that, however, the compilation unit would have to be a range sequence, with the range elements being the `SingleTypeImportDeclaration`s and the `TypeDeclarationsOpt`. I'm not certain whether it is possible for a single symbol to inherit *both* `RangeSequence` and `RangeMulInh`, and I don't want to get distracted by that issue (which is irrelevant for type analysis).

The scope of a type imported by a single-type-import declaration or type-import-on-demand declaration is all of the class and interface type declarations in the compilation unit in which the import declaration appears.

Scopes[11]:

```
TREE SYMBOL CompilationUnit INHERITS TypExportRange COMPUTE
  SYNT.TypScopeKey=NoKey;
END;

RULE: CompilationUnit ::=
        PackageDeclarationOpt
        ImportJavaLang ImportDeclarationsOpt TypeDeclarationsOpt
COMPUTE
  CompilationUnit.TypGotLocKeys=
    ImportJavaLang.TypGotLocKeys
    <- ImportDeclarationsOpt CONSTITUENTS (
      TypeImportOnDemandDeclaration.TypGotLocKeys,
      SingleTypeImportDeclaration.TypGotLocKeys);
END;

TREE SYMBOL ImportJavaLang INHERITS PkgIdUseEnv END;

RULE: ImportJavaLang ::= COMPUTE
  ImportJavaLang.Sym=MakeName("java.lang");
  ImportJavaLang.TypGotLocKeys=
    AddPackage(
      GetPtyScope(ImportJavaLang.PkgKey,NoEnv),
      INCLUDING CompilationUnit.TypEnv);
END;

TREE SYMBOL QualInhName INHERITS PkgIdUseEnv END;

RULE: TypeImportOnDemandDeclaration ::= 'import' QualInhName '.' '*' COMPUTE
  TypeImportOnDemandDeclaration.TypGotLocKeys=
    AddPackage(
      GetPtyScope(QualInhName.PkgKey,NoEnv),
      INCLUDING CompilationUnit.TypEnv);
```

```
END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

Each of the donating environments must be created from information about the specified packages and types. We must ensure that all of the members of all packages have been defined. Those computations are specific to the context of the donating environment:

Scopes[12]:

```
TREE SYMBOL Goal COMPUTE
  SYNT.TypEnv = TypRootEnv <- CONSTITUENTS PtyAnyScope.PtyGotLocKeys;
END;

TREE SYMBOL TypeDeclarationsOpt INHERITS TypExportRange COMPUTE
  INH.TypEnv=AddPackage(THIS.PtyEnv,NewScope(INCLUDING TypAnyScope.TypEnv));
END;

RULE: CompilationUnit ::=
        PackageDeclarationOpt
        ImportJavaLang ImportDeclarationsOpt TypeDeclarationsOpt
COMPUTE
  TypeDeclarationsOpt.TypScopeKey=PackageDeclarationOpt.PkgKey;
END;

RULE: SingleTypeImportDeclaration ::= 'import' QualInhName COMPUTE
  SingleTypeImportDeclaration.TypGotLocKeys=
    BindKeyInScope(
      INCLUDING CompilationUnit.TypEnv,
      HeadintList(QualInhName.Ids),
      KeyInScope(TypRootEnv, QualInhName.Sym));
END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

FIXME: If the identifier is already bound in the environment, `BindKeyInScope` returns `NoBinding` and does not re-bind the identifier. Depending on the particular environment used, that condition might indicate a multiple definition error that should be reported.

FIXME: Depending on the situation, we might want to check modifiers of the type being bound.

Environment AddPackage(Environment pkg, Environment env)[13]:

```
/* Add the types in a package to an environment
 *   On entry-
 *     pkg=environment of the package
 *     env=environment to which types are to be added
 *   On exit-
 *     All visible types of package sym have been added to env
 ***/
{ Binding bind;
```

```
  for (bind = DefinitionsOf(pkg);
       bind != NoBinding;
       bind = NextDefinition(bind)) {
    BindKeyInScope(env, IdnOf(bind), KeyOf(bind));
  }

  return env;
}
```

This macro is invoked in definition 39.

4.2.2 Class and interface declarations

The scope of a type introduced by a class type declaration or interface type declaration is the declarations of all class and interface types in all the compilation units of the package in which it is declared. Class and interface declarations are children of the `TypeDeclarationsOpt` child of the `CompilationUnit` node. Thus each `TypeDeclarationsOpt` node provides the appropriate scope for these declarations.

`TypIdDefScope` computations normally create a new definition table key to be bound to the identifier. In this case, however, we want to bind the key for the fully-qualified type identifier to the simple type name being defined. Therefore we need to override the normal binding computation:

Scopes[14]:

```
ATTR IsLocalClass: int;

TREE SYMBOL Goal COMPUTE SYNT.IsLocalClass=0; END;

CLASS SYMBOL LocalTypeRange INHERITS TypExportRange COMPUTE
  SYNT.TypScopeKey=NoKey;
  SYNT.IsLocalClass=1;
END;

TREE SYMBOL MethodDeclaration    INHERITS LocalTypeRange END;
TREE SYMBOL Block                INHERITS LocalTypeRange END;
TREE SYMBOL SwitchBlockStatement INHERITS LocalTypeRange END;

TREE SYMBOL TypeIdDef INHERITS IdentOcc, TypIdDefScope, MultDefChk END;

RULE: TypeDeclaration ::=
        Modifiers 'class' TypeIdDef Super Interfaces ClassBody  COMPUTE
  TypeDeclaration.Ids=
    ConsintList(TypeIdDef.Sym,INCLUDING CompilationUnit.Ids);
END;

RULE: TypeDeclaration ::=
        Modifiers 'interface' TypeIdDef Interfaces InterfaceBody COMPUTE
  TypeDeclaration.Ids=
    ConsintList(TypeIdDef.Sym,INCLUDING CompilationUnit.Ids);
END;
```

```
RULE: CompilationUnit ::=
        PackageDeclarationOpt
        ImportJavaLang ImportDeclarationsOpt TypeDeclarationsOpt
COMPUTE
  CompilationUnit.Ids=PackageDeclarationOpt.Ids;
END;

RULE: PackageDeclarationOpt ::= 'package' PackageName ';' COMPUTE
  PackageDeclarationOpt.Ids=PackageName.Ids;
END;

RULE: PackageDeclarationOpt ::= COMPUTE
  PackageDeclarationOpt.Ids=SingleintList(0);
END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

4.2.3 Member declarations

The scope of a member declared or inherited by a class type or interface type is the entire declaration of the class or interface type. A class or interface may also inherit members from other classes or interfaces that it extends:

Scopes[15]:

```
ATTR TypScopeKey, FldScopeKey, MthScopeKey: DefTableKey;
ATTR TypEnv, FldEnv, MthEnv: Environment;

TREE SYMBOL ClassBody     INHERITS TypeBody END;
TREE SYMBOL InterfaceBody INHERITS TypeBody END;

CLASS SYMBOL TypeBody INHERITS TypExportInhRange, FldExportInhRange,
                               MthExportInhRange END;

RULE: TypeDeclaration ::=
        Modifiers 'class' TypeIdDef Super Interfaces ClassBody  COMPUTE
  ClassBody.TypScopeKey=TypeDeclaration.Type;
  ClassBody.TypGotInh=CONSTITUENTS InheritMembers.TypInheritOk;
  TypeDeclaration.TypEnv=ClassBody.TypEnv;
  TypeDeclaration.TypScopeKey=TypeDeclaration.Type;

  ClassBody.FldScopeKey=TypeDeclaration.Type;
  ClassBody.FldGotInh=CONSTITUENTS InheritMembers.FldInheritOk;
  TypeDeclaration.FldEnv=ClassBody.FldEnv;
  TypeDeclaration.FldScopeKey=TypeDeclaration.Type;

  ClassBody.MthScopeKey=TypeDeclaration.Type;
  ClassBody.MthGotInh=CONSTITUENTS InheritMembers.MthInheritOk;
  TypeDeclaration.MthEnv=ClassBody.MthEnv;
  TypeDeclaration.MthScopeKey=TypeDeclaration.Type;
```

```
END;

RULE: TypeDeclaration ::=
        Modifiers 'interface' TypeIdDef Interfaces InterfaceBody COMPUTE
  InterfaceBody.TypScopeKey=TypeDeclaration.Type;
  InterfaceBody.TypGotInh=1;/* FIXME CONSTITUENTS
  InheritMembers.TypInheritOk; */
  TypeDeclaration.TypEnv=InterfaceBody.TypEnv;
  TypeDeclaration.TypScopeKey=TypeDeclaration.Type;

  InterfaceBody.FldScopeKey=TypeDeclaration.Type;
  InterfaceBody.FldGotInh=CONSTITUENTS InheritMembers.FldInheritOk;
  TypeDeclaration.FldEnv=InterfaceBody.FldEnv;
  TypeDeclaration.FldScopeKey=TypeDeclaration.Type;

  InterfaceBody.MthScopeKey=TypeDeclaration.Type;
  InterfaceBody.MthGotInh=CONSTITUENTS InheritMembers.MthInheritOk;
  TypeDeclaration.MthEnv=InterfaceBody.MthEnv;
  TypeDeclaration.MthScopeKey=TypeDeclaration.Type;
END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

Array types have a `length` field that is not explicitly declared:

Scopes[16]:

```
TREE SYMBOL ArrayType INHERITS FldExportRange COMPUTE
  INH.FldScopeKey=
    ORDER(
      ResetTypeOf(DefineIdn(THIS.FldEnv,MakeName("length")),intType),
      THIS.Type);
END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

The inheritances for classes and interfaces can be handled in the same way:

Scopes[17]:

```
CLASS SYMBOL InheritMembers INHERITS FldInheritScope, MthInheritScope,
                                     TypInheritScope
COMPUTE
  INH.FldInnerScope=INCLUDING TypeDeclaration.FldEnv;
  INH.MthInnerScope=INCLUDING TypeDeclaration.MthEnv;
  INH.TypInnerScope=INCLUDING TypeDeclaration.TypEnv;
  IF(AND(
      NE(INCLUDING TypeDeclaration.Type,objectType),
      OR(
        NOT(THIS.TypInheritOk),
        OR(NOT(THIS.FldInheritOk),NOT(THIS.MthInheritOk)))),
```

```
    message(ERROR,"Improper inheritance",0,COORDREF));
END;

TREE SYMBOL Super INHERITS InheritMembers COMPUTE
  SYNT.TypScopeKey=THIS.Type;
  SYNT.FldScopeKey=THIS.Type;
  SYNT.MthScopeKey=THIS.Type;
END;

RULE: Super ::= 'extends' InhName COMPUTE
  Super.Type=InhName.Type;
END;

RULE: Super ::= COMPUTE
  Super.Type=objectType;
END;

TREE SYMBOL InterfaceType INHERITS InheritMembers END;

RULE: InterfaceType ::= InhName COMPUTE
  InterfaceType.TypScopeKey=InhName.Type;
  InterfaceType.FldScopeKey=InhName.Type;
  InterfaceType.MthScopeKey=InhName.Type;
  InterfaceType.Sym=InhName.Sym;
END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

Neither fields nor methods have fully-qualified names. Thus there is no need to override the module's computation of their bindings. Method identifiers can be overloaded, however, and therefore we don't want to report an error if one has multiple defining occurrences:

Scopes[18]:

```
TREE SYMBOL FieldIdDef  INHERITS IdentOcc, FldIdDefScope, MultDefChk END;
TREE SYMBOL MethodIdDef INHERITS IdentOcc, MthIdDefScope             END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

4.2.4 Anonymous classes

Scopes[19]:

```
TREE SYMBOL TypeIdUse INHERITS IdentOcc, TypQualIdUse END;

RULE: Expression ::=
  Expression '.' 'new' TypeIdUse '(' Arguments ')' AnonymousClass
COMPUTE
```

```
  TypeIdUse.TypScopeKey=Expression[2].Type;
  TypeIdUse.Key=TypeIdUse.TypKey;
END;

SYMBOL AnonymousClass INHERITS TypRangeScope, FldRangeScope, MthRangeScope
END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

4.2.5 Parameter and variable declarations

The scope of a parameter is the entire body of the method, constructor or exception handler in which it is declared. The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration. The scope of a local variable declared in the `ForInit` part of a `for` statement includes its own initializer, any further declarators to the right in the `ForInit` part of the `for` statement, the `Expression` and `ForUpdate` parts of the `for` statement, and the contained `Statement`. The scope of the parameter of a `CatchClause` is the `Block` of that clause.

Scopes[20]:

```
TREE SYMBOL MethodDeclaration      INHERITS VarRangeScope END;
TREE SYMBOL ConstructorDeclaration INHERITS VarRangeScope END;
TREE SYMBOL Block                  INHERITS VarRangeScope END;
TREE SYMBOL ForStatement           INHERITS VarRangeScope END;
TREE SYMBOL CatchClause            INHERITS VarRangeScope END;
```

This macro is defined in definitions 11, 12, 14, 15, 16, 17, 18, 19, and 20.
This macro is invoked in definition 2.

Although a parameter actually obeys Algol scope rules rather than C scope rules, its declaration occurs syntactically at the beginning of the construct that is its scope. In this case, the Algol scope rules and C scope rules have the same effect. Therefore we can use the Eli `CScope` module to implement the scope rules for parameters. This simplifies the analysis of applied occurrences, because parameters and local variables are disambiguated in the same way.

4.3 Applied occurrences of names

Applied occurrences of names are classified according to their syntactic context. Each may be a single identifier (a *simple name*) or a sequence of two or more identifiers, separated by dots (a *qualified name*).

Applied occurrences of names[21]:

Type names[22]
Expression names[23]
Method names[24]
Ambiguous names[25]
Qualified names[26]

This macro is defined in definitions 21, 31, 32, 33, and 34.
This macro is invoked in definition 2.

4.3.1 Type names

Type names[22]:

```
RULE: TypeName ::= Name $pTypeName COMPUTE
  TypeName.Key=pTypeName.Key;
END;

TREE SYMBOL sTypeIdUse    INHERITS TypIdUseEnv,  IdentOcc END;
TREE SYMBOL qTypeIdUse    INHERITS TypQualIdUse, IdentOcc END;
TREE SYMBOL pPkgOrTypName INHERITS PkgIdUseEnv COMPUTE
  SYNT.Sym=FullyQualifiedName(THIS.Ids);
END;
TREE SYMBOL pTypeName COMPUTE
  SYNT.Sym=FullyQualifiedName(THIS.Ids);
END;

RULE: pTypeName ::= sTypeIdUse COMPUTE
  pTypeName.Ids=SingleintList(sTypeIdUse.Sym);
  pTypeName.Key=sTypeIdUse.TypKey;
END;

RULE: pTypeName ::= pPkgOrTypName qTypeIdUse COMPUTE
  pTypeName.Ids=ConsintList(qTypeIdUse.Sym,pPkgOrTypName.Ids);
  pTypeName.Key=qTypeIdUse.TypKey;
  qTypeIdUse.TypScopeKey=TransDefer(pPkgOrTypName.TypScopeKey);
END;

RULE: pPkgOrTypName ::= sTypeIdUse COMPUTE
  pPkgOrTypName.Ids=SingleintList(sTypeIdUse.Sym);
  pPkgOrTypName.TypScopeKey=
    IF(NE(sTypeIdUse.TypKey,NoKey),TransDefer(sTypeIdUse.TypKey),
    pPkgOrTypName.PkgKey);
END;

RULE: pPkgOrTypName ::= pPkgOrTypName qTypeIdUse COMPUTE
  pPkgOrTypName[1].Ids=ConsintList(qTypeIdUse.Sym,pPkgOrTypName[2].Ids);
  pPkgOrTypName[1].TypScopeKey=
    IF(NE(qTypeIdUse.TypKey,NoKey),TransDefer(qTypeIdUse.TypKey),
    pPkgOrTypName[1].PkgKey);
  qTypeIdUse.TypScopeKey=pPkgOrTypName[2].TypScopeKey;
END;
```

This macro is defined in definitions 22.
This macro is invoked in definition 21.

4.3.2 Expression names

Expression names[23]:

```
RULE: Expression ::= Name $pExpressionName COMPUTE
END;
```

```
TREE SYMBOL sExprIdUse    INHERITS VarIdUseEnv,  FldIdUseEnv,  IdentOcc END;
TREE SYMBOL qExprIdUse    INHERITS               FldQualIdUse, IdentOcc END;
TREE SYMBOL pExpressionName COMPUTE
  SYNT.Sym=FullyQualifiedName(THIS.Ids);
END;

RULE: pExpressionName ::= COMPUTE
  pExpressionName.Ids=SingleintList(NoStrIndex);
  pExpressionName.Key=NoKey;
END;

RULE: pExpressionName ::= sExprIdUse COMPUTE
  pExpressionName.Ids=SingleintList(sExprIdUse.Sym);
  pExpressionName.Key=
    IF(NE(sExprIdUse.VarKey,NoKey),sExprIdUse.VarKey,
    IF(NE(sExprIdUse.FldKey,NoKey),sExprIdUse.FldKey,
    NoKey));
END;

RULE: pExpressionName ::= pAmbiguousName qExprIdUse COMPUTE
  pExpressionName.Ids=ConsintList(qExprIdUse.Sym,pAmbiguousName.Ids);
  pExpressionName.Key=qExprIdUse.FldKey;
  qExprIdUse.FldScopeKey=pAmbiguousName.TypScopeKey;
END;
```

This macro is defined in definitions 23.
This macro is invoked in definition 21.

4.3.3 Method names

Method names[24]:

```
RULE: MethodName ::= Name $pMethodName COMPUTE
  MethodName.Key=pMethodName.Key;
END;

TREE SYMBOL sMethIdUse    INHERITS MthIdUseEnv,  IdentOcc END;
TREE SYMBOL qMethIdUse    INHERITS MthQualIdUse, IdentOcc END;
TREE SYMBOL pMethodName COMPUTE
  SYNT.Sym=FullyQualifiedName(THIS.Ids);
END;

RULE: pMethodName ::= sMethIdUse COMPUTE
  pMethodName.Ids=SingleintList(sMethIdUse.Sym);
  pMethodName.Key=sMethIdUse.MthKey;
END;

RULE: pMethodName ::= pAmbiguousName qMethIdUse COMPUTE
  pMethodName.Ids=ConsintList(qMethIdUse.Sym,pAmbiguousName.Ids);
  pMethodName.Key=qMethIdUse.MthKey;
  qMethIdUse.MthScopeKey=pAmbiguousName.TypScopeKey;
END;
```

This macro is invoked in definition 21.

4.3.4 Ambiguous names

Ambiguous names[25]:

```
TREE SYMBOL sAmbgIdUse     INHERITS VarIdUseEnv,
                                    FldIdUseEnv,  TypIdUseEnv,  IdentOcc END;
TREE SYMBOL qAmbgIdUse     INHERITS FldQualIdUse, TypQualIdUse, IdentOcc END;
TREE SYMBOL pAmbiguousName INHERITS PkgIdUseEnv COMPUTE
  SYNT.Sym=FullyQualifiedName(THIS.Ids);
END;

RULE: pAmbiguousName ::= sAmbgIdUse COMPUTE
  pAmbiguousName.Ids=SingleintList(sAmbgIdUse.Sym);
  pAmbiguousName.TypScopeKey=
    IF(NE(sAmbgIdUse.VarKey,NoKey),
      TransDefer(GetTypeOf(sAmbgIdUse.VarKey,NoKey)),
    IF(NE(sAmbgIdUse.FldKey,NoKey),
      TransDefer(GetTypeOf(sAmbgIdUse.FldKey,NoKey)),
    IF(NE(sAmbgIdUse.TypKey,NoKey),
      TransDefer(sAmbgIdUse.TypKey),
    pAmbiguousName.PkgKey)));
END;

RULE: pAmbiguousName ::= pAmbiguousName qAmbgIdUse COMPUTE
  pAmbiguousName[1].Ids=ConsintList(qAmbgIdUse.Sym,pAmbiguousName[2].Ids);
  pAmbiguousName[1].TypScopeKey=
    IF(NE(qAmbgIdUse.FldKey,NoKey),
      TransDefer(GetTypeOf(qAmbgIdUse.FldKey,NoKey)),
    IF(NE(qAmbgIdUse.TypKey,NoKey),
      TransDefer(qAmbgIdUse.TypKey),
    pAmbiguousName[1].PkgKey));
  qAmbgIdUse.TypScopeKey=pAmbiguousName[2].TypScopeKey;
  qAmbgIdUse.FldScopeKey=pAmbiguousName[2].TypScopeKey;
END;
```

This macro is invoked in definition 21.

4.3.5 Qualified names

All qualified names are represented by the single nonterminal `Name`.

In some contexts, an initial segment of the qualified name is a package name. We consider a package name to be a single symbol consisting of the intial segment of the qualified name including the separating dots. Thus we must be able to convert any initial segment of a qualified name to a single symbol. Our strategy is to accumulate the identifiers making up the qualified name as a list of string table indices in reverse order:

Qualified names[26]:

```
ATTR Ids: intList;

TREE SYMBOL InhBaseId INHERITS IdentOcc END;
```

```
RULE: QualInhName ::= InhBaseId COMPUTE
  QualInhName.Ids=SingleintList(InhBaseId.Sym);
END;

TREE SYMBOL InhQualId INHERITS IdentOcc END;

RULE: QualInhName ::= QualInhName '.' InhQualId COMPUTE
  QualInhName[1].Ids=ConsintList(InhQualId.Sym,QualInhName[2].Ids);
END;
```

This macro is invoked in definition 21.

Instantiate required modules[27]:

```
$/Adt/List.gnrc +instance=int :inst
```

This macro is defined in definitions 3, 6, 7, 27, 30, and 36.
This macro is invoked in definition 1.

Any sublist of this list easily be converted to a string by a recursive function that reassembles the components in the proper order:

int FullyQualifiedName(intList rep)[28]:

```
/* Convert a qualified name to a single symbol
 *   On entry-
 *     rep=list representing the initial segment to be converted
 *   On exit-
 *     FullyQualifiedName=string table index of the symbol
 ***/
{ if (rep) {
    MakeFQName(rep);
    obstack_1grow(Csm_obstk, '\0');
    CsmStrPtr = (char *)obstack_finish(Csm_obstk);
    return MakeName(CsmStrPtr);
  } else
    return 0;
}
```

This macro is invoked in definition 39.

void MakeFQName(intList rep)[29]:

```
/* Build a string by reversing a list
 *   On entry-
 *     rep=list to be reversed and converted
 *   On exit-
 *     Csm_obstk contains the converted string, unterminated
 ***/
{ char *temp;

  intList tail = TailintList(rep);
```

```
  if (tail != NULLintList) {
    MakeFQName(tail);
    obstack_1grow(Csm_obstk, '.');
  }
  temp = StringTable(HeadintList(rep));
  obstack_grow(Csm_obstk, temp, strlen(temp));
}
```

This macro is invoked in definition 39.

Instantiate required modules[30]:

```
$/Tech/MakeName.gnrc +instance=Identifier :inst
```

This macro is defined in definitions 3, 6, 7, 27, 30, and 36.
This macro is invoked in definition 1.

Applied occurrences of names[31]:

```
ATTR Key: DefTableKey;

RULE: PackageName ::= QualInhName COMPUTE
  PackageName.Ids=QualInhName.Ids;
  PackageName.Sym=QualInhName.Sym;
END;

RULE: InhName ::= QualInhName COMPUTE
  InhName.Ids=QualInhName.Ids;
  InhName.Sym=QualInhName.Sym;
  InhName.Key=QualInhName.Key;
  InhName.Type=TransDefer(QualInhName.Type);
END;
```

This macro is defined in definitions 21, 31, 32, 33, and 34.
This macro is invoked in definition 2.

4.3.6 Determining the meaning of a name

Applied occurrences of names[32]:

```
ATTR Sym: int;

TREE SYMBOL QualInhName INHERITS PkgIdUseEnv, TypeDefUseId COMPUTE
  SYNT.Sym=FullyQualifiedName(THIS.Ids);
END;

TREE SYMBOL InhBaseId INHERITS TypIdUseEnv END;

RULE: QualInhName ::= InhBaseId COMPUTE
  QualInhName.Key=InhBaseId.TypKey;
  QualInhName.TypScopeKey=
    IF(NE(InhBaseId.TypKey,NoKey),TransDefer(InhBaseId.TypKey),
```

```
    QualInhName.PkgKey);
END;

TREE SYMBOL InhQualId INHERITS TypQualIdUse COMPUTE
  SYNT.TypBind=
    BindingInScope(THIS.TypScope,THIS.Sym)
    <- (INCLUDING TypAnyScope.TypGotVisibleKeysNest);
END;

RULE: QualInhName ::= QualInhName '.' InhQualId COMPUTE
  QualInhName[1].Key=InhQualId.TypKey;
  QualInhName[1].TypScopeKey=
    IF(NE(InhQualId.TypKey,NoKey),TransDefer(InhQualId.TypKey),
    QualInhName[1].PkgKey);
  InhQualId.TypScopeKey=QualInhName[2].TypScopeKey;
END;
```

This macro is defined in definitions 21, 31, 32, 33, and 34.
This macro is invoked in definition 2.

4.3.7 Field access

Applied occurrences of names[33]:

```
SYMBOL FieldIdUse INHERITS IdentOcc, FldQualIdUse COMPUTE
  SYNT.Key=THIS.FldKey;
END;

RULE: Expression ::= Expression '.' FieldIdUse COMPUTE
  FieldIdUse.MthScopeKey=Expression[2].Type;
END;

RULE: Expression ::= 'super' '.' FieldIdUse COMPUTE
  FieldIdUse.MthScopeKey=FinalType(INCLUDING TypeDeclaration.SuperType);
END;

RULE: Expression ::= Name $pTypeName '.' 'super' '.' FieldIdUse COMPUTE
  FieldIdUse.MthScopeKey=
    FinalType(GetSuperType(FinalType(pTypeName.Type),NoKey));
END;
```

This macro is defined in definitions 21, 31, 32, 33, and 34.
This macro is invoked in definition 2.

4.3.8 Method access

Applied occurrences of names[34]:

```
SYMBOL MethodIdUse INHERITS IdentOcc, MthQualIdUse COMPUTE
  SYNT.Key=THIS.MthKey;
END;
```

```
RULE: Expression ::= 'super' '.' MethodIdUse '(' Arguments ')' COMPUTE
  MethodIdUse.MthScopeKey=FinalType(INCLUDING TypeDeclaration.Type);
END;

RULE: Expression ::= Expression '.' MethodIdUse '(' Arguments ')' COMPUTE
  MethodIdUse.MthScopeKey=Expression[2].Type;
END;
```

This macro is defined in definitions 21, 31, 32, 33, and 34.
This macro is invoked in definition 2.

4.4 Labeled Statements

Labeled Statements[35]:

```
TREE SYMBOL LabeledStatement INHERITS LblRangeScope END;
TREE SYMBOL LabelIdDef       INHERITS IdentOcc, LblIdDefScope END;
TREE SYMBOL LabelIdUse       INHERITS IdentOcc, LblIdUseEnv, LblChkIdUse END;
```

This macro is invoked in definition 2.

Instantiate required modules[36]:

```
$/Name/CScope.gnrc +instance=Lbl +referto=Lbl :inst
```

This macro is defined in definitions 3, 6, 7, 27, 30, and 36.
This macro is invoked in definition 1.

4.5 Support code

Name.h[37]:

```
#ifndef NAME_H
#define NAME_H

#include "eliproto.h"
#include "envmod.h"
#include "intList.h"

extern Environment AddPackage ELI_ARG((Environment, Environment));
extern int FullyQualifiedName ELI_ARG((intList));

#endif
```

This macro is attached to a product file.

Name.head[38]:

```
#include "Name.h"
```

This macro is attached to a product file.

A type-c file implements the operations and data structures of the abstract data types:

Name.c[39]:

```
#include "pdl_gen.h"
#include "Strings.h"
#include "MakeName.h"
#include "PkgAlgScope.h"
#include "Name.h"

Environment
#if PROTO_OK
AddPackage(Environment pkg, Environment env)
#else
AddPackage(pkg, env) Environment pkg, env;
#endif
```
Environment AddPackage(Environment pkg, Environment env)[13]

```
static void
#if PROTO_OK
MakeFQName(intList rep)
#else
MakeFQName(rep) intList rep;
#endif
```
void MakeFQName(intList rep)[29]

```
int
#if PROTO_OK
FullyQualifiedName(intList rep)
#else
FullyQualifiedName(rep) intList rep;
#endif
```
int FullyQualifiedName(intList rep)[28]

This macro is attached to a product file.

Chapter 5

Type Analysis

Java type analysis is complex, but it is based upon a number of language-independent concepts. Eli type analysis modules encapsulate the necessary computations and the dependence relationships among them, and export a nomenclature for language constructs. Type analyzer code for Java can be created by instantiating the modules and then classifying Java constructs according to that nomenclature.

Type.specs[1]:

Instantiate required modules[15]

This macro is attached to a product file.

Section 5.1 defines the Java type model, and the necessary type denotations and type identifiers are implemented in Section 5.2. Identifiers representing typed entities (fields, variables, and parameters) are established in Section 5.3.

Language-defined operators and the mechanisms for creating methods are defined in Section 5.4. The chapter concludes with definitions of type relationships in expressions (Section 5.6) and statements (Section 5.5).

Type.lido[2]:

```
ATTR Type: DefTableKey;
```

Types and type identifiers[14]
Typed identifiers[34]
Method declaration[39]
Expressions[45]
Statements[42]

This macro is attached to a product file.

The process described in this chapter associates a type with every construct yielding a value and every identifier representing a type or typed entity. It relies not only on computations in the abstract syntax tree, but also on properties stored in the definition table.

Type.pdl[3]:

Property definitions[18]

This macro is attached to a product file.

5.1 The Java type model

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

Type.oil[4]:

Primitive types[5]
Reference types[9]
void[13]

This macro is attached to a product file.

5.1.1 Integral types

Java defines some operators only for integral types. Note that the shift operators do not demand that both operands be of the same type.

Primitive types[5]:

```
SET integralType  = [byteType, shortType, intType, longType, charType];
SET integrl2Type  = integralType;

OPER
  lshiftOp, rshiftOp, urshiftOp(integralType,integrl2Type):integralType;
  complOp(integralType):integralType;
  andOp, orOp, exorOp(integralType,integralType):integralType;

INDICATION
  llInd:    lshiftOp;
  ggInd:    rshiftOp;
  gggInd:   urshiftOp;
  tildeInd: complOp;
  ampInd:   andOp;
  barInd:   orOp;
  upInd:    exorOp;

COERCION
  (byteType): charType;
  (byteType): shortType;
  (shortType): intType;
  (intType): longType;
  (charType): intType;
```

This macro is defined in definitions 5, 6, 7, and 8.
This macro is invoked in definition 4.

5.1.2 Numeric types

Arithmetic operations are defined on both integral and floating-point operands. There are no operators that operate exclusively on floating-point operands.

Primitive types[6]:

```
SET floatingType = [floatType, doubleType];
SET numericType  = integralType + floatingType;
SET numerc2Type  = numericType;

OPER
  cmplsOp, cmpgtOp, cmpleOp, cmpgeOp(numericType,numericType): boolType;
  incrOp, decrOp, posOp, negOp(numericType): numericType;
  addOp, subOp, mulOp, divOp, remOp(numericType,numericType): numericType;
  castNumOp(numericType):numerc2Type;

INDICATION
  leqInd:        cmpleOp;
  lssInd:        cmplsOp;
  geqInd:        cmpgeOp;
  gtrInd:        cmpgtOp;
  plusplusInd:   incrOp;
  minusminusInd: decrOp;
  plusInd:       posOp, addOp;
  minusInd:      negOp, subOp;
  starInd:       mulOp;
  slashInd:      divOp;
  percentInd:    remOp;
  castInd:       castNumOp;

COERCION
  (longType): floatType;
  (floatType): doubleType;
```

This macro is defined in definitions 5, 6, 7, and 8.
This macro is invoked in definition 4.

5.1.3 boolean type

Boolean operations include the normal logical operators and also conditional versions of disjunction and conjunction.

Primitive types[7]:

```
OPER
  invOp(boolType): boolType;
  disjOp, conjOp, candOp, corOp(boolType,boolType): boolType;

INDICATION
  bangInd:   invOp;
  barbarInd: corOp;
  ampampInd: candOp;
```

This macro is defined in definitions 5, 6, 7, and 8.
This macro is invoked in definition 4.

5.1.4 Primitive types

Tests for equality are available for all primitive types, and also for string types. Also, a value of any primitive type can be concatenated with a string to yield a string.

Primitive types[8]:

```
SET primitiveType = [boolType] + numericType;

OPER
  cmpeqOp, cmpneOp(primitiveType,primitiveType): boolType;
  prmCondOp(primitiveType,primitiveType): primitiveType;
  prmstrOp(primitiveType,stringType): stringType;
  strprmOp(stringType,primitiveType): stringType;

INDICATION
  eqlInd:  cmpeqOp;
  neqInd:  cmpneOp;
  plusInd: strprmOp, prmstrOp;
  conditionalInd: prmCondOp;
  equalInd: prmCondOp;
```

This macro is defined in definitions 5, 6, 7, and 8.
This macro is invoked in definition 4.

5.1.5 Classes

Every user-defined class is a subclass of a parent class (which may be java.lang.Object). A value of the parent class can be cast to the type of an object, an object can be concatenated to a string, its type can be queried, and references can be compared for equality.

An OIL class is used as a template to create the operations required by the Java class:

Reference types[9]:

```
CLASS classOps() BEGIN
  OPER
    strclsOp(classOps,stringType): stringType;
    clsstrOp(stringType,classOps): stringType;
    clseqOp, clsneOp(classOps,classOps): boolType;
    clsCondOp(classOps,classOps): classOps;

  COERCION
    (nullType): classOps;
END;

CLASS classInh(parentClass) BEGIN
  OPER
    narrowOp(parentClass): classInh;

  COERCION
    (classInh): parentClass;
END;

INDICATION
  plusInd:        strclsOp, clsstrOp;
  castInd:        narrowOp;
  conditionalInd: clsCondOp;
  eqlInd:         clseqOp;
```

```
  neqInd:         clsneOp;
  equalInd:       clsCondOp;
```

This macro is defined in definitions 9, 10, 11, and 12.
This macro is invoked in definition 4.

A special type is used as the type of a class literal, and we need a bogus operator to define it:

Reference types[10]:

```
OPER classOp(classType): classType;
```

This macro is defined in definitions 9, 10, 11, and 12.
This macro is invoked in definition 4.

5.1.6 Throwable

We need a bogus operator to define `throwableType` as a type:

Reference types[11]:

```
OPER throwOp(throwableType): throwableType;
```

This macro is defined in definitions 9, 10, 11, and 12.
This macro is invoked in definition 4.

5.1.7 Arrays

Reference types[12]:

```
CLASS arrayOps(elementType) BEGIN
  OPER
    arrayinit(elementType): arrayOps;
    arrayaccess(arrayOps,intType): elementType;
    obj2arrOp(objectType): arrayOps;
    strarrOp(arrayOps,stringType): stringType;
    arrstrOp(stringType,arrayOps): stringType;
    arrCondOp(arrayOps,arrayOps): arrayOps;
    arreqOp, arrneOp(arrayOps,arrayOps): boolType;

  COERCION
    (arrayOps): objectType;
    (nullType): arrayOps;
END;

INDICATION
  arrayInit:      arrayinit;
  arrayAccess:    arrayaccess;
  plusInd:        strarrOp, arrstrOp;
  castInd:        obj2arrOp;
  conditionalInd: arrCondOp;
  eqlInd:         arreqOp;
  neqInd:         arrneOp;
  equalInd:       arrCondOp;
```

This macro is defined in definitions 9, 10, 11, and 12.
This macro is invoked in definition 4.

5.1.8 void

Void is not a type in Java. Nevertheless, a method that does not return a result is declared using the keyword `void` in place of the result type. In order to allow a uniform representation of methods, it is useful to define a "void" type as the result type in those cases. Since this fictitious type does not participate in any normal Java operations, we define a bogus operator in order to get OIL to regard `voidType` as a legitimate type:

void[13]:

```
OPER voidOp(voidType): voidType;
```

This macro is invoked in definition 4.

5.2 Types and type identifiers

Java distinguishes primitive types, class and interface types, and array types. Primitive types are language-defined and named by keywords. Class and interface types are program-defined and named by type names. Array types are anonymous, and can only be denoted by using brackets.

Types and type identifiers[14]:

```
ATTR Type: DefTableKey;

RULE: Type ::= PrimitiveType COMPUTE
  Type.Type=PrimitiveType.Type;
END;
```

PrimitiveType[16]

```
RULE: Type ::= TypeName COMPUTE
  Type.Type=TypeName.Type;
END;
```

Class and interface types[17]

```
RULE: Type ::= ArrayType COMPUTE
  Type.Type=ArrayType.Type;
END;
```

Array types[28]

```
RULE: Type ::= 'void' COMPUTE
  Type.Type=voidType;
END;
```

This macro is invoked in definition 2.

Chapter 4 showed how each type name and typed entity name had its `Key` attribute set to the appropriate definition table key. Thus the `Typing` module should be instantiated without a `referto` parameter:

Instantiate required modules[15]:

```
$/Type/Typing.gnrc :inst
```

This macro is defined in definitions 15, 26, 38, and 44.
This macro is invoked in definition 1.

5.2.1 Primitive types

Primitive types defined by Java are represented in the text of a program by keywords:

PrimitiveType[16]:

```
RULE: PrimitiveType ::= 'boolean' COMPUTE
  PrimitiveType.Type=boolType;
END;

RULE: PrimitiveType ::= 'byte' COMPUTE
  PrimitiveType.Type=byteType;
END;

RULE: PrimitiveType ::= 'short' COMPUTE
  PrimitiveType.Type=shortType;
END;

RULE: PrimitiveType ::= 'int' COMPUTE
  PrimitiveType.Type=intType;
END;

RULE: PrimitiveType ::= 'long' COMPUTE
  PrimitiveType.Type=longType;
END;

RULE: PrimitiveType ::= 'char' COMPUTE
  PrimitiveType.Type=charType;
END;

RULE: PrimitiveType ::= 'float' COMPUTE
  PrimitiveType.Type=floatType;
END;

RULE: PrimitiveType ::= 'double' COMPUTE
  PrimitiveType.Type=doubleType;
END;
```

This macro is invoked in definition 14.

5.2.2 Class and interface types

Each class or interface type is declared once and named, and then represented by its name wherever it is used in the program text. No two class or interface types are equivalent, even though they may have identical declarations:

Class and interface types[17]:

```
SYMBOL TypeIdDef       INHERITS TypeDefDefId   END;
SYMBOL TypeName        INHERITS TypeDefUseId   END;
SYMBOL pTypeName       INHERITS TypeDefUseId   END;

RULE: TypeDeclaration ::=
```

```
        Modifiers 'class' TypeIdDef Super Interfaces ClassBody
COMPUTE
  TypeIdDef.Type=TypeDeclaration.Type;
END;

RULE: TypeDeclaration ::=
        Modifiers 'interface' TypeIdDef Interfaces InterfaceBody
COMPUTE
  TypeIdDef.Type=TypeDeclaration.Type;
END;
```

This macro is defined in definitions 17, 19, 20, 21, 22, 23, and 25.
This macro is invoked in definition 14.

Interface types differ in several ways from class types, and therefore we use a property to distinguish them:

Property definitions[18]:

```
IsInterfaceType: int;
```

This macro is defined in definitions 18, 24, 27, and 29.
This macro is invoked in definition 3.

Class and interface types[19]:

```
RULE: TypeDeclaration ::=
        Modifiers 'interface' TypeIdDef Interfaces InterfaceBody
COMPUTE
  TypeDeclaration.GotType=ResetIsInterfaceType(TypeDeclaration.Type,1);
END;
```

This macro is defined in definitions 17, 19, 20, 21, 22, 23, and 25.
This macro is invoked in definition 14.

The `TypeDenotation` role establishes the value of the `Type` attribute as a new definition table key, but three classes in the `java.lang` package (`Object`, `String`, and `Throwable`) are represented in this specification by known keys. Therefore it is necessary to override the computation of the `Type` attribute in a class declaration:

Class and interface types[20]:

```
SYMBOL TypeDeclaration INHERITS TypeDenotation END;

RULE: TypeDeclaration ::=
        Modifiers 'class' TypeIdDef Super Interfaces ClassBody COMPUTE
  TypeDeclaration.Type=
    IF(NOT(INCLUDING CompilationUnit.IsJavaLang),NewType(),
    IF(EQ(TypeIdDef.Sym,MakeName("Object")),objectType,
    IF(EQ(TypeIdDef.Sym,MakeName("String")),stringType,
    IF(EQ(TypeIdDef.Sym,MakeName("Throwable")),throwableType,
    NewType()))));
END;
```

```
ATTR IsJavaLang: int;

RULE: CompilationUnit ::=
        PackageDeclarationOpt
        ImportJavaLang ImportDeclarationsOpt TypeDeclarationsOpt
COMPUTE
  CompilationUnit.IsJavaLang=PackageDeclarationOpt.IsJavaLang;
END;

RULE: PackageDeclarationOpt ::= 'package' PackageName ';' COMPUTE
  PackageDeclarationOpt.IsJavaLang=
    EQ(FullyQualifiedName(PackageName.Ids),MakeName("java.lang"));
END;

RULE: PackageDeclarationOpt ::= COMPUTE
  PackageDeclarationOpt.IsJavaLang=0;
END;
```

This macro is defined in definitions 17, 19, 20, 21, 22, 23, and 25.
This macro is invoked in definition 14.

Each class and interface type has associated operators, defined by instantiating the OIL class `classOps`:

Class and interface types[21]:

```
SYMBOL TypeDeclaration INHERITS OperatorDefs END;

TREE SYMBOL TypeDeclaration COMPUTE
  SYNT.GotOper=
    ORDER(
      InstClass0(classOps,THIS.Type),
      ListOperator(
        FinalType(THIS.Type),
        NoOprName,
        NULLDefTableKeyList,
        THIS.Type));
END;
```

This macro is defined in definitions 17, 19, 20, 21, 22, 23, and 25.
This macro is invoked in definition 14.

There are also operators associated with inheritance, defined by instantiating `classInh`:

Class and interface types[22]:

```
SYMBOL Super INHERITS OperatorDefs END;

RULE: Super ::= 'extends' InhName COMPUTE
  Super.GotOper=
    InstClass1(classInh,INCLUDING TypeDeclaration.Type,InhName.Type);
END;

RULE: Super ::= COMPUTE
```

```
  Super.GotOper=
    InstClass1(classInh,INCLUDING TypeDeclaration.Type,objectType);
END;

SYMBOL InterfaceType INHERITS OperatorDefs END;

RULE: InterfaceType ::= InhName COMPUTE
  InterfaceType.GotOper=
    InstClass1(classInh,INCLUDING TypeDeclaration.Type,InhName.Type);
END;
```

This macro is defined in definitions 17, 19, 20, 21, 22, 23, and 25.
This macro is invoked in definition 14.

We need to keep track of the supertype of each class in order to access it when the keyword `super` shows up in expressions:

Class and interface types[23]:

```
TREE SYMBOL TypeDeclaration: SuperType: DefTableKey;

RULE: TypeDeclaration ::=
        Modifiers 'class' TypeIdDef Super Interfaces ClassBody COMPUTE
  TypeDeclaration.SuperType=Super.Type;
END;

RULE: TypeDeclaration ::=
        Modifiers 'interface' TypeIdDef Interfaces InterfaceBody COMPUTE
  TypeDeclaration.SuperType=NoKey;
END;
```

This macro is defined in definitions 17, 19, 20, 21, 22, 23, and 25.
This macro is invoked in definition 14.

Not all of the contexts in which the supertype is needed are textually enclosed by the type in question. Therefore we need to store the supertype as a property of the type, and ensure that it has been set whenever it is accessed.

Property definitions[24]:

```
SuperType: DefTableKey;
```

This macro is defined in definitions 18, 24, 27, and 29.
This macro is invoked in definition 3.

Class and interface types[25]:

```
TREE SYMBOL Goal COMPUTE
  SYNT.GotSuperTypes=CONSTITUENTS TypeDeclaration.GotSuperType;
END;

TREE SYMBOL TypeDeclaration COMPUTE
  SYNT.GotSuperType=ResetSuperType(THIS.Type,THIS.SuperType);
END;
```

This macro is defined in definitions 17, 19, 20, 21, 22, 23, and 25.
This macro is invoked in definition 14.

5.2.3 Array types

Array types cannot be declared or named; every use of an array type is represented by its denotation. Two array type denotations are equivalent if they have the same element type and the same number of dimensions. This is a restricted form of structural equivalence, which requires that the `StructEquiv` module be instantiated:

Instantiate required modules[26]:

```
$/Type/StructEquiv.fw
```

This macro is defined in definitions 15, 26, 38, and 44.
This macro is invoked in definition 1.

`StructEquiv` defines a partition on the set of types such that types that could be equivalent on the basis of their construction rule lie in the same block of the partition. After all type denotations have been examined, `StructEquiv` refines the partition on the basis of the component types of each type. The blocks of the refined partitions are the type equivalence classes.

We therefore use `AddTypeToBlock` to assign each array denotation to the `ArrayTypes` block with the element type as a singleton component type list:

Property definitions[27]:

```
ArrayTypes;
```

This macro is defined in definitions 18, 24, 27, and 29.
This macro is invoked in definition 3.

Array types[28]:

```
TREE SYMBOL ArrayType: ElementType: DefTableKey;

TREE SYMBOL ArrayType INHERITS TypeDenotation COMPUTE
  SYNT.GotType=
    AddTypeToBlock(
      THIS.Type,
      ArrayTypes,
      SingleDefTableKeyList(THIS.ElementType));
END;

RULE: ArrayType ::= PrimitiveType '[' ']' COMPUTE
  ArrayType.ElementType=PrimitiveType.Type;
END;

RULE: ArrayType ::= Name $pTypeName '[' ']' COMPUTE
  ArrayType.ElementType=pTypeName.Type;
END;

RULE: ArrayType ::= ArrayType '[' ']' COMPUTE
  ArrayType[1].ElementType=ArrayType[2].Type;
END;
```

This macro is defined in definitions 28, 30, 31, 32, and 33.
This macro is invoked in definition 14.

Each array type has an associated set of operators that must be added to the operator database. Those operators should only be instantiated once for each unique array type. The `Ops` property of the final array type is set to indicate that the operators for that array type have been entered into the database.

Property definitions[29]:

```
Ops: int [Has, KReset];
```

This macro is defined in definitions 18, 24, 27, and 29.
This macro is invoked in definition 3.

Array types[30]:

```
TREE SYMBOL ArrayType INHERITS OperatorDefs COMPUTE
  SYNT.GotOper=
    IF(NOT(HasOps(THIS.Type)),
      InstClass1(
        arrayOps,
        KResetOps(FinalType(THIS.Type),1),
        THIS.ElementType));
END;
```

This macro is defined in definitions 28, 30, 31, 32, and 33.
This macro is invoked in definition 14.

Every array access is also an array type denotation, but with some subscripts specified. The processing of these denotations is basically the same as the processing specified above. The only difference is that there is no direct access to the element type. We therefore use a chain, started in the context of the array access, to carry the element type from one dimension to the next:

Array types[31]:

```
CHAIN ArrTyp: DefTableKey;

SYMBOL Dimension INHERITS TypeDenotation, OperatorDefs COMPUTE
  SYNT.GotType=
    AddTypeToBlock(THIS.Type,ArrayTypes,SingleDefTableKeyList(THIS.ArrTyp));
  SYNT.GotOper=
    IF(NOT(HasOps(THIS.Type)),
      InstClass1(
        arrayOps,
        KResetOps(FinalType(THIS.Type),1),
        THIS.ArrTyp));
  THIS.ArrTyp=THIS.Type;
END;
```

This macro is defined in definitions 28, 30, 31, 32, and 33.
This macro is invoked in definition 14.

As discussed in Section 5.3, Java allows identifiers to be declared with trailing dimensions. This requires operations like those above, associated with the particular context in which the trailing dimensions occur. If there is no trailing dimension, do nothing:

Array types[32]:

```
TREE SYMBOL VariableDeclaratorId INHERITS TypeDenotation, OperatorDefs
COMPUTE
  SYNT.GotType="yes";
  SYNT.GotOper="yes";
END;

RULE: VariableDeclaratorId ::= VariableDeclaratorId '[' ']' COMPUTE
  VariableDeclaratorId[1].GotType=
    AddTypeToBlock(
      VariableDeclaratorId[1].Type,
      ArrayTypes,
      SingleDefTableKeyList(
        INCLUDING (VariableDeclaratorId.Type,TypedDefinition.Type)));
  VariableDeclaratorId[1].GotOper=
    IF(NOT(HasOps(VariableDeclaratorId[1].Type)),
      InstClass1(
        arrayOps,
        KResetOps(FinalType(VariableDeclaratorId[1].Type),1),
        INCLUDING (VariableDeclaratorId.Type,TypedDefinition.Type)));
END;
```

This macro is defined in definitions 28, 30, 31, 32, and 33.
This macro is invoked in definition 14.

For compatibility with older versions of the Java platform, a declaration form for a method that returns an array is allowed to place (some or all of) the empty bracket pairs that form the declaration of the array type after the parameter list. By now, the operations should be familiar:

Array types[33]:

```
TREE SYMBOL MethodDeclarator INHERITS TypeDenotation, OperatorDefs
COMPUTE
  SYNT.GotType="yes";
  SYNT.GotOper="yes";
END;

RULE: MethodDeclarator ::= MethodDeclarator '[' ']' COMPUTE
  MethodDeclarator[1].GotType=
    AddTypeToBlock(
      MethodDeclarator[1].Type,
      ArrayTypes,
      SingleDefTableKeyList(
        INCLUDING (MethodDeclarator.Type,MethodHeader.Type)));
  MethodDeclarator[1].GotOper=
    IF(NOT(HasOps(MethodDeclarator[1].Type)),
      InstClass1(
        arrayOps,
        KResetOps(FinalType(MethodDeclarator[1].Type),1),
        INCLUDING (MethodDeclarator.Type,MethodHeader.Type)));
END;
```

This macro is defined in definitions 28, 30, 31, 32, and 33.
This macro is invoked in definition 14.

See Section 5.4.2 for the context of this computation.

5.3 Typed identifiers

Fields, local variables, and formal parameters are the only typed identifiers in Java. Formal parameters are considered to be variables, and their defining occurrences are represented by the nonterminal `VariableDeclaratorId`.

Typed identifiers[34]:

```
TREE SYMBOL FieldDeclarators    INHERITS TypedDefinition END;
TREE SYMBOL VariableDeclarators INHERITS TypedDefinition END;
TREE SYMBOL FormalParameter     INHERITS TypedDefinition END;

TREE SYMBOL FieldIdDef          INHERITS TypedDefId      END;
TREE SYMBOL VariableIdDef       INHERITS TypedDefId      END;
```

This macro is defined in definitions 34, 35, and 36.
This macro is invoked in definition 2.

All of the typed definitions get their type information from their context, and the simple identifier definitions reach up to those typed definitions.

Typed identifiers[35]:

```
RULE: FieldDeclaration ::= Modifiers Type FieldDeclarators ';' COMPUTE
  FieldDeclarators.Type=Type.Type;
END;

RULE: LocalVariableDeclaration ::= 'final' Type VariableDeclarators COMPUTE
  VariableDeclarators.Type=Type.Type;
END;

RULE: LocalVariableDeclaration ::= Type VariableDeclarators COMPUTE
  VariableDeclarators.Type=Type.Type;
END;

RULE: FormalParameter ::= 'final' Type VariableDeclaratorId COMPUTE
  FormalParameter.Type=Type.Type;
END;

RULE: FormalParameter ::= Type VariableDeclaratorId COMPUTE
  FormalParameter.Type=Type.Type;
END;
```

This macro is defined in definitions 34, 35, and 36.
This macro is invoked in definition 2.

Java also allows a C-like declaration in which the dimensions follow the identifier. In fact, it is legal to mix the two. We implement this requirement by using the `VariableDeclaratorId` as a mechanism to provide the necessary type denotations (see Section 5.2.3 for the necessary definitions). If no dimension is attached to the variable, then the type is passed through unmodified:

Typed identifiers[36]:

```
TREE SYMBOL VariableDeclaratorId COMPUTE
  SYNT.Type=INCLUDING TypedDefinition.Type;
END;

RULE: VariableDeclaratorId ::= VariableIdDef COMPUTE
  VariableIdDef.Type=VariableDeclaratorId.Type;
END;
```

This macro is defined in definitions 34, 35, and 36.
This macro is invoked in definition 2.

5.4 Indications

An indication is a set of operators with different signatures. It is represented by a definition table key. Some indications are defined by the language, and often represented by sequences of special characters. Java methods are also indications, which are defined by the user.

The specification must associate the indication's definition table key with each occurrence of a source language representation of that indication.

5.4.1 Language-defined indications

All language-defined indications are represented by known keys defined in Section 5.1. The set of operators associated with the indication was also defined there.

This Section assigns the same indication to an `Operator` and to the related `AssignmentOperator`. The reason is that the type analysis of assignment operations is split into two parts: the assignment part and the operation part. The operation part is processed in exactly the same manner as a dyadic operation with the related `Operator`.

Operator.d[37]:

```
PreDefInd('^=',   AssignmentOperator, upInd)
PreDefInd('<<=',  AssignmentOperator, llInd)
PreDefInd('=',    AssignmentOperator, equalInd)
PreDefInd('>>=',  AssignmentOperator, ggInd)
PreDefInd('>>>=', AssignmentOperator, gggInd)
PreDefInd('|=',   AssignmentOperator, barInd)
PreDefInd('-=',   AssignmentOperator, minusInd)
PreDefInd('/=',   AssignmentOperator, slashInd)
PreDefInd('*=',   AssignmentOperator, starInd)
PreDefInd('&=',   AssignmentOperator, ampInd)
PreDefInd('%=',   AssignmentOperator, percentInd)
PreDefInd('+=',   AssignmentOperator, plusInd)
PreDefInd('^',              Operator, upInd)
PreDefInd('<<',             Operator, llInd)
PreDefInd('<=',             Operator, leqInd)
PreDefInd('<',              Operator, lssInd)
PreDefInd('==',             Operator, eqlInd)
PreDefInd('>=',             Operator, geqInd)
PreDefInd('>>>',            Operator, gggInd)
PreDefInd('>>',             Operator, ggInd)
PreDefInd('>',              Operator, gtrInd)
PreDefInd('||',             Operator, barbarInd)
```

```
PreDefInd('|',              Operator, barInd)
PreDefInd('+',              Operator, plusInd)
PreDefInd('-',              Operator, minusInd)
PreDefInd('!=',             Operator, neqInd)
PreDefInd('/',              Operator, slashInd)
PreDefInd('*',              Operator, starInd)
PreDefInd('&',              Operator, ampInd)
PreDefInd('&&',             Operator, ampampInd)
PreDefInd('%',              Operator, percentInd)
PreDefInd('++',             Operator, plusplusInd)
PreDefInd('--',             Operator, minusminusInd)
PreDefInd('~',              Operator, tildeInd)
PreDefInd('!',              Operator, bangInd)
```

This macro is attached to a non-product file.

These definitions are implemented by the `PreDefOp` module, which reads the non-product file `Operator.d`:

Instantiate required modules[38]:

```
$/Type/PreDefOp.gnrc +referto=(Operator.d) :inst
```

This macro is defined in definitions 15, 26, 38, and 44.
This macro is invoked in definition 1.

5.4.2 Method declaration

Each Java method declaration defines one operator in the method indication's set. Name analysis (Chapter 4) associates a definition table key with the method name, and that key is used as the indication. Each use of the method name is an instance of the `MethodName` node.

Method declaration[39]:

```
RULE: MethodHeader ::= Modifiers Type MethodDeclarator Throws COMPUTE
  MethodHeader.Type=Type.Type;
END;

TREE SYMBOL MethodDeclarator COMPUTE
  SYNT.Type=INCLUDING MethodHeader.Type;
END;

RULE: MethodDeclarator ::= MethodIdDef '(' FormalParameters ')' COMPUTE
  MethodDeclarator.GotOper=
    ListOperator(
      MethodIdDef.Key,
      NoOprName,
      FormalParameters.OpndTypeList,
      MethodDeclarator.Type);
END;
```

This macro is defined in definitions 39, 40, and 41.
This macro is invoked in definition 2.

The roundabout mechanism for transmitting the return type of the method is due to the obsolete requirement for allowing dimensions to follow the parameter list when the method's return type is an array Section 5.2.3 contains additional computations associated with `MethodDeclarator` in this case.

Computational roles provide all of the necessary computations for gathering up the formal parameter types and computing the `OpndTypeList` attribute:

Method declaration[40]:

```
SYMBOL FormalParameters INHERITS OpndTypeListRoot END;
SYMBOL FormalParameter  INHERITS OpndTypeListElem END;
```

This macro is defined in definitions 39, 40, and 41.
This macro is invoked in definition 2.

A constructor declaration is very similar to a method declaration, except that the type name serves as the method name:

Method declaration[41]:

```
SYMBOL ConstructorDeclaration INHERITS OperatorDefs END;

RULE: ConstructorDeclaration ::=
        Modifiers TypeName '(' FormalParameters ')' Throws
        '{' Statements '}' COMPUTE
  ConstructorDeclaration.GotOper=
    ListOperator(
      TypeName.TypeKey,
      NoOprName,
      FormalParameters.OpndTypeList,
      TypeName.TypeKey);
END;
```

This macro is defined in definitions 39, 40, and 41.
This macro is invoked in definition 2.

5.5 Statements

If a statement represents a context in which an expression is required to yield a specific type, `Expression.Required` must be set to that type. Computation provided by the type analysis modules will then report an error if the actual type is not acceptable as the required type.

Statements[42]:

```
RULE: Statement ::=
        'if' '(' Expression ')' Statement 'else' Statement COMPUTE
  Expression.Required=boolType;
END;

RULE: Statement ::= 'if' '(' Expression ')' Statement COMPUTE
  Expression.Required=boolType;
END;

RULE: WhileStatement ::= 'while' '(' Expression ')' Statement COMPUTE
```

```
  Expression.Required=boolType;
END;

RULE: DoStatement ::= 'do' Statement 'while' '(' Expression ')' ';' COMPUTE
  Expression.Required=boolType;
END;

RULE: ForTest ::= Expression COMPUTE
  Expression.Required=boolType;
END;

RULE: Statement ::= 'throw' Expression ';' COMPUTE
  Expression.Required=throwableType;
END;
```

This macro is defined in definitions 42 and 43.
This macro is invoked in definition 2.

The expression of a return statement must return a value compatible with the method containing the return statement. This requires an extra attribute at the appropriate point to specify that type. Constructors and class initializers cannot return values, so the void type is specified in those cases. Since no other type is acceptable as the void type, this will result in an error report if a return statement with an expression is used in a constructor or class initializer.

Statements[43]:

```
ATTR ReturnType: DefTableKey;

RULE: Statement ::= 'return' Expression ';' COMPUTE
  Expression.Required=
    INCLUDING (MethodBody.ReturnType,
               ConstructorDeclaration.ReturnType,
               ClassInitializer.ReturnType);
END;

RULE: MethodDeclarator ::= MethodIdDef '(' FormalParameters ')' COMPUTE
  MethodIdDef.ReturnType=MethodDeclarator.Type;
END;

RULE: MethodDeclaration ::= MethodHeader MethodBody COMPUTE
  MethodBody.ReturnType=MethodHeader CONSTITUENT MethodIdDef.ReturnType;
END;

SYMBOL ConstructorDeclaration COMPUTE
  SYNT.ReturnType=voidType;
END;

SYMBOL ClassInitializer COMPUTE
  SYNT.ReturnType=voidType;
END;
```

This macro is defined in definitions 42 and 43.
This macro is invoked in definition 2.

The convoluted mechanism for obtaining the method return type is again due to the obsolete construct discussed in Section 5.2.3.

5.6 Expressions

Java overloading can be resolved on the basis of operand types alone, and therefore the `Expression` module should be instantiated without a `referto` parameter:

Instantiate required modules[44]:

```
$/Type/Expression.gnrc :inst
```

This macro is defined in definitions 15, 26, 38, and 44.
This macro is invoked in definition 1.

Expressions[45]:

```
SYMBOL Expression INHERITS ExpressionSymbol END;
```

Lexical literals[46]
Class literal[47]
this[48]
Class instance creation[49]
Array creation[50]
Field access[51]
Method invocation[52]
Array access[53]
Names[54]
Operators other than assignment[55]
Cast expressions[56]
Conditional expressions[57]
Assignment operators[58]

This macro is invoked in definition 2.

5.6.1 Lexical literals

A literal denotes a fixed, unchanging value. Its type is defined by the language for each kind of literal. All literals are primary contexts:

Lexical literals[46]:

```
RULE: Expression ::= CharacterLiteral COMPUTE
  PrimaryContext(Expression,charType);
END;

RULE: Expression ::= IntLiteral COMPUTE
  PrimaryContext(Expression,intType);
END;

RULE: Expression ::= LongLiteral COMPUTE
  PrimaryContext(Expression,longType);
END;
```

```
RULE: Expression ::= FloatLiteral COMPUTE
  PrimaryContext(Expression,floatType);
END;

RULE: Expression ::= DoubleLiteral COMPUTE
  PrimaryContext(Expression,doubleType);
END;

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

RULE: Expression ::= 'true' COMPUTE
  PrimaryContext(Expression,boolType);
END;

RULE: Expression ::= 'null' COMPUTE
  PrimaryContext(Expression,nullType);
END;

RULE: Expression ::= StringLiteral COMPUTE
  PrimaryContext(Expression,stringType);
END;
```

This macro is invoked in definition 45.

5.6.2 Class literal

The type of a class literal is `classType`.

Class literal[47]:

```
RULE: Expression ::= Name $pTypeName '.' 'class' COMPUTE
  PrimaryContext(Expression,classType);
END;
```

This macro is invoked in definition 45.

5.6.3 this

The keyword `this` represents the current instance of a class whose definition lexically encloses the expression. If the keyword is unqualified, then the closest-containing class definition provides the result type:

this[48]:

```
RULE: Expression ::= 'this' COMPUTE
  PrimaryContext(Expression,INCLUDING TypeDeclaration.Type);
END;

RULE: Expression ::= Name $pTypeName '.' 'this' COMPUTE
  PrimaryContext(Expression,pTypeName.Type);
END;
```

This macro is invoked in definition 45.

5.6.4 Class instance creation

The name following the `new` specifies a constructor that should be applied to the argument list to create a value.

Class instance creation[49]:

```
TREE SYMBOL TypeIdUse INHERITS TypeDefUseId END;

RULE: Expression ::=
  Expression '.' 'new' TypeIdUse '(' Arguments ')' AnonymousClass
COMPUTE
  ListContext(Expression[1],,Arguments);
  Indication(FinalType(TypeIdUse.Type));
END;

RULE: Expression ::= 'new' TypeName '(' Arguments ')' AnonymousClass
COMPUTE
  ListContext(Expression,,Arguments);
  Indication(FinalType(TypeName.Type));
END;
```

This macro is invoked in definition 45.

5.6.5 Array creation

Array creation[50]:

```
RULE: Expression ::= 'new' TypeName Dimensions COMPUTE
  CHAINSTART Dimensions.ArrTyp=TypeName.Type;
  PrimaryContext(Expression,Dimensions.ArrTyp);
END;

RULE: Expression ::= 'new' PrimitiveType Dimensions COMPUTE
  CHAINSTART Dimensions.ArrTyp=PrimitiveType.Type;
  PrimaryContext(Expression,Dimensions.ArrTyp);
END;

RULE: Dimension ::= '[' Expression ']' COMPUTE
  Expression.Required=longType;
END;
```

This macro is invoked in definition 45.

5.6.6 Field access

Field access[51]:

```
SYMBOL FieldIdUse INHERITS TypedUseId END;

RULE: Expression ::= Expression '.' FieldIdUse COMPUTE
  PrimaryContext(Expression[1],FieldIdUse.Type);
END;
```

```
RULE: Expression ::= 'super' '.' FieldIdUse COMPUTE
  PrimaryContext(Expression,FieldIdUse.Type);
END;

RULE: Expression ::= Name $pTypeName '.' 'super' '.' FieldIdUse COMPUTE
  PrimaryContext(Expression,FieldIdUse.Type);
END;
```

This macro is invoked in definition 45.

5.6.7 Method invocation

Method invocation[52]:

```
RULE: Expression ::= MethodName '(' Arguments ')' COMPUTE
  ListContext(Expression,,Arguments);
  Indication(MethodName.Key);
END;

RULE: Expression ::= 'super' '.' MethodIdUse '(' Arguments ')' COMPUTE
  ListContext(Expression,,Arguments);
  Indication(MethodIdUse.Key);
END;

RULE: Expression ::= Expression '.' MethodIdUse '(' Arguments ')' COMPUTE
  ListContext(Expression[1],,Arguments);
  Indication(MethodIdUse.Key);
END;

SYMBOL Arguments INHERITS OpndExprListRoot END;
SYMBOL Argument  INHERITS OpndExprListElem END;

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

TREE SYMBOL ExpressionStatement INHERITS ExpressionSymbol END;

RULE: ExpressionStatement ::= 'super' '(' Arguments ')' ';' COMPUTE
  ListContext(ExpressionStatement,,Arguments);
  Indication(INCLUDING TypeDeclaration.SuperType);
END;

RULE: ExpressionStatement ::= 'this' '(' Arguments ')' ';' COMPUTE
  ListContext(ExpressionStatement,,Arguments);
  Indication(INCLUDING TypeDeclaration.Type);
  IF(AND(NOT(BadIndication),BadOperator),
    message(ERROR,"Invalid constructor call",0,COORDREF));
END;
```

This macro is invoked in definition 45.

5.6.8 Array access

Array access[53]:

```
RULE: Expression ::= Expression '[' Expression ']' COMPUTE
  DyadicContext(Expression[1],,Expression[2],Expression[3]);
  Indication(arrayAccess);
END;
```

This macro is invoked in definition 45.

5.6.9 Names

Names[54]:

```
TREE SYMBOL pExpressionName INHERITS TypedUseId END;

RULE: Expression ::= Name $pExpressionName COMPUTE
  PrimaryContext(Expression,pExpressionName.Type);
END;
```

This macro is invoked in definition 45.

5.6.10 Operators other than assignment

Operators other than assignment[55]:

```
SYMBOL Operator INHERITS OperatorSymbol END;

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

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

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

This macro is invoked in definition 45.

5.6.11 Cast expressions

Cast expressions[56]:

```
RULE: Expression ::= '(' PrimitiveType ')' Expression COMPUTE
  PrimaryContext(Expression[1],PrimitiveType.Type);
  RootContext(PrimitiveType.Type,,Expression[2]);
  Indication(castInd);
END;

RULE: Expression ::= '(' ArrayType ')' Expression COMPUTE
```

```
  PrimaryContext(Expression[1],ArrayType.Type);
  RootContext(ArrayType.Type,,Expression[2]);
  Indication(castInd);
END;

RULE: Expression ::= '(' Expression $pTypeName ')' Expression COMPUTE
  PrimaryContext(Expression[1],pTypeName.Key);
END;
```

This macro is invoked in definition 45.

Conditional expressions[57]:

```
RULE: Expression ::= Expression '?' Expression ':' Expression COMPUTE
  Expression[2].Required=boolType;
  DyadicContext(Expression[1],,Expression[3],Expression[4]);
  Indication(conditionalInd);
END;

RULE: Expression ::= Expression 'instanceof' Type COMPUTE
  PrimaryContext(Expression[1],boolType);
END;
```

This macro is invoked in definition 45.

5.6.12 Assignment operators

Assignment operators[58]:

```
SYMBOL LeftHandSide       INHERITS ExpressionSymbol END;
SYMBOL RightHandSide      INHERITS ExpressionSymbol END;
SYMBOL AssignmentOperator INHERITS OperatorSymbol   END;

RULE: Expression ::= LeftHandSide AssignmentOperator RightHandSide COMPUTE
  DyadicContext(Expression,AssignmentOperator,LeftHandSide,RightHandSide);
END;

RULE: RightHandSide ::= Expression COMPUTE
  ConversionContext(RightHandSide,,Expression);
  Indication(castInd);  /* FIXME: Verify that castInd is correct */
END;
```

This macro is invoked in definition 45.

Chapter 6

Check Context Conditions

Context.lido[1]:

```
#define IsReferenceType(x)\
  OR(OR(EQ(x,classType),GetIsInterfaceType(x,0)),IsCoercible(x,objectType))
```

Types, values, and variables[4]
Names[5]
Classes[11]
Blocks and statements[14]
Expressions[26]

This macro is attached to a product file.

Context.specs[2]:

Instantiate required modules[7]

This macro is attached to a product file.

Context.pdl[3]:

Property definitions[10]

This macro is attached to a product file.

6.1 Types, values, and variables

Types, values, and variables[4]:

This macro is invoked in definition 1.

6.2 Names

Names[5]:

```
TREE SYMBOL TypeIdDef INHERITS ChkTypeDefDefId END;

RULE: TypeName ::= Name $pTypeName COMPUTE
```

```
  TypeName.Sym=pTypeName.Sym;
  TypeName.Type=pTypeName.Type;
END;

TREE SYMBOL TypeName INHERITS ChkTypeDefUseId COMPUTE
  IF(EQ(THIS.Type,NoKey),
    message(
      ERROR,
      CatStrInd("Invalid type name: ",THIS.Sym),
      0,
      COORDREF));
END;

RULE: Expression ::= Name $pExpressionName COMPUTE
  IF(AND(
      NE(pExpressionName.Sym,NoStrIndex),
      EQ(pExpressionName.Key,NoKey)),
    message(
      ERROR,
      CatStrInd(
        "Invalid variable or field name: ",
        pExpressionName.Sym),
      0,
      COORDREF));
END;

RULE: MethodName ::= Name $pMethodName COMPUTE
  MethodName.Sym=pMethodName.Sym;
END;

TREE SYMBOL MethodName COMPUTE
  IF(EQ(THIS.Key,NoKey),
    message(
      ERROR,
      CatStrInd("Invalid method name: ",THIS.Sym),
      0,
      COORDREF))
    <- INCLUDING Goal.GotAllConstants;
END;

TREE SYMBOL InhName COMPUTE
  IF(EQ(THIS.Key,NoKey),
    message(
      ERROR,
      CatStrInd("Invalid Type name: ",THIS.Sym),
      0,
      COORDREF))
    <- INCLUDING Goal.GotAllConstants;
END;

RULE: SingleTypeImportDeclaration ::= 'import' QualInhName COMPUTE
```

```
  IF(EQ(QualInhName.Key,NoKey),
    message(
      ERROR,
      CatStrInd("Invalid Type name: ",QualInhName.Sym),
      0,
      COORDREF))
    <- INCLUDING Goal.GotAllConstants;
END;
```

Multiple definition errors[6]

This macro is invoked in definition 1.

6.2.1 Multiple definition errors

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

Multiple definition errors[6]:

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

This macro is defined in definitions 6 and 8.
This macro is invoked in definition 5.

`MultDefChk` requires the Eli `Unique` module, and the error reporting requires the string concatenation module:

Instantiate required modules[7]:

```
$/Prop/Unique.gnrc :inst
$/Tech/Strings.specs
```

This macro is defined in definitions 7 and 19.
This macro is invoked in definition 2.

Type, field, and variable identifiers can have only one defining occurrence in their scope. Method identifiers, on the other hand, can be overloaded and therefore may have several defining occurrences.

Multiple definition errors[8]:

```
TREE SYMBOL TypeIdDef     INHERITS IdentOcc, TypIdDefScope, MultDefChk COMPUTE
  SYNT.Key=THIS.TypKey;
END;
```

```
TREE SYMBOL FieldIdDef    INHERITS IdentOcc, FldIdDefScope, MultDefChk COMPUTE
  SYNT.Key=THIS.FldKey;
END;

TREE SYMBOL VariableIdDef INHERITS IdentOcc, VarIdDefScope, MultDefChk COMPUTE
  SYNT.Key=THIS.VarKey;
END;

TREE SYMBOL MethodIdDef   INHERITS IdentOcc, MthIdDefScope             COMPUTE
  SYNT.Key=THIS.MthKey;
END;
```

This macro is defined in definitions 6 and 8.
This macro is invoked in definition 5.

The definition of a legal assignment can be found in Section 5.2 of the Java specification.

int NoAssign(DefTableKey from, DefTableKey to, int v)[9]:

```
/* Determine whether assignment is legal
 *   On entry-
 *     from=type of the value
 *     to=type of the variable
 *     v=constant
 *   If the assignment is legal then on exit-
 *     NoAssign=0
 *   Else on exit-
 *     NoAssign=1
 ***/
{ if (IsCoercible(from, to)) return 0;
  if (IsCoercible(from, intType) && IsCoercible(to, intType)) {
    char *vs;

    if (v == NoStrIndex) return 1;
    vs = strsub(GetUpperBound(to, "0"),StringTable(v),10);
    if (vs[0] == '-') return 1;
    vs = strsub(StringTable(v),GetLowerBound(to, "0"),10);
    if (vs[0] == '-') return 1;
    return 0;
  }
  return 1;
}
```

This macro is invoked in definition 43.

Property definitions[10]:

```
LowerBound, UpperBound: CharPtr;

byteType  -> LowerBound={"-128"},   UpperBound={"127"};
shortType -> LowerBound={"-32768"}, UpperBound={"32767"};
charType  -> LowerBound={"0"},      UpperBound={"65535"};
```

This macro is defined in definitions 10, 16, 22, 36, and 40.
This macro is invoked in definition 3.

6.3 Classes

Classes[11]:

Initializers[13]
Method throws[12]

This macro is invoked in definition 1.

6.3.1 Method throws

Method throws[12]:

```
RULE: ThrownType ::= TypeName COMPUTE
  IF(NOT(IsCoercible(FinalType(TypeName.Type),throwableType)),
    message(
     ERROR,
     CatStrInd("Must be throwable: ",TypeName.Sym),
     0,
     COORDREF))
    <- INCLUDING Goal.GotAllConstants;
END;
```

This macro is invoked in definition 11.

6.3.2 Initializers

Initializers[13]:

```
ATTR ElementType: DefTableKey;
ATTR ComponentTypes: DefTableKeyList;

RULE: Initializer ::= Expression COMPUTE
  RootContext(Initializer.ElementType,,Expression);
  Indication(castInd);
  IF(NoAssign(Expression.Type,Initializer.ElementType,Expression.Constant),
    message(ERROR,"Incorrect type for this context",0,COORDREF));
END;

RULE: Initializer ::= '{' Initializers '}' COMPUTE
  .ComponentTypes=
    GetComponentTypes(Initializer.ElementType,NULLDefTableKeyList);
  Initializers.ElementType=
    IF(EQ(.ComponentTypes,NULLDefTableKeyList),
      NoKey,
      HeadDefTableKeyList(.ComponentTypes));
  IF(AND(
      NE(Initializer.ElementType,NoKey),
      EQ(Initializers.ElementType,NoKey)),
    message(ERROR,"Invalid initializer for this array",0,COORDREF));
END;

RULE: InitialElement ::= Initializer COMPUTE
```

```
  Initializer.ElementType=INCLUDING Initializers.ElementType;
END;

RULE: FieldDeclarator ::= FieldDeclaratorId '=' Initializer COMPUTE
  Initializer.ElementType=FieldDeclaratorId.ElementType;
END;

RULE: VariableDeclarator ::= VariableDeclaratorId '=' Initializer COMPUTE
  Initializer.ElementType=VariableDeclaratorId.ElementType;
END;

RULE: FieldDeclaratorId ::= FieldIdDef COMPUTE
  FieldDeclaratorId.ElementType=FieldIdDef.Type;
END;

RULE: VariableDeclaratorId ::= VariableDeclaratorId '[' ']' COMPUTE
  VariableDeclaratorId[1].ElementType=VariableDeclaratorId[2].ElementType;
END;

RULE: VariableDeclaratorId ::= VariableIdDef COMPUTE
  VariableDeclaratorId.ElementType=VariableIdDef.Type;
END;
```

This macro is invoked in definition 11.

6.4 Blocks and statements

Blocks and statements[14]:

Labeled statements[15]
The switch statement[17]
The break statement[20]
The continue statement[21]
The return statement[23]
The synchronized statement[24]
The try statement[25]

This macro is invoked in definition 1.

6.4.1 Labeled statements

Labeled statements[15]:

```
ATTR LblDepth: int;
CLASS SYMBOL LabelTree COMPUTE
  SYNT.LblDepth=ADD(INCLUDING LabelTree.LblDepth,1);
END;

TREE SYMBOL MethodDeclaration      INHERITS LabelTree END;
TREE SYMBOL ConstructorDeclaration INHERITS LabelTree END;
TREE SYMBOL ClassInitializer       INHERITS LabelTree END;
TREE SYMBOL Goal                   INHERITS LabelTree COMPUTE
```

```
  SYNT.LblDepth=0;
  SYNT.GotLblDepth="yes";
END;

TREE SYMBOL LabeledStatement: Sym: int;

TREE SYMBOL LabeledStatement INHERITS LblIdUseEnv COMPUTE
  IF(EQ(GetLblDepth(THIS.LblKey,0),INCLUDING LabelTree.LblDepth),
    message(
      ERROR,
      CatStrInd("Shadows another label: ",THIS.Sym),
      0,
      COORDREF))
  <- INCLUDING (LabeledStatement.GotLblDepth,Goal.GotLblDepth);
END;

RULE: LabeledStatement ::= LabelIdDef ':' Statement COMPUTE
  LabeledStatement.Sym=LabelIdDef.Sym;
  LabeledStatement.GotLblDepth=
    ResetLblDepth(LabelIdDef.LblKey,INCLUDING LabelTree.LblDepth);
END;
```

This macro is invoked in definition 14.

Property definitions[16]:

```
LblDepth: int;
```

This macro is defined in definitions 10, 16, 22, 36, and 40.
This macro is invoked in definition 3.

6.4.2 The switch statement

The switch statement[17]:

```
SYMBOL SwitchStatement INHERITS CaseRangeScope END;
SYMBOL SwitchStatement: SwitchExpType: DefTableKey;

RULE: SwitchStatement ::= 'switch' '(' Expression ')' SwitchBlock COMPUTE
  SwitchStatement.SwitchExpType=Expression.Type;
  Expression.Required=intType;
END;

RULE: SwitchLabel ::= 'case' Expression ':' COMPUTE
  IF(EQ(Expression.Constant,NoStrIndex),
    message(ERROR,"Selector must be a constant",0,COORDREF),
  IF(NoAssign(
      Expression.Type,
      INCLUDING SwitchStatement.SwitchExpType,
      Expression.Constant),
    message(ERROR,"Selector not assignable to switch",0,COORDREF)))
  <- INCLUDING Goal.GotAllConstants;
END;
```

This macro is defined in definitions 17 and 18.
This macro is invoked in definition 14.

The switch statement[18]:

```
SYMBOL SwitchLabel INHERITS CaseIdDefScope, MultDefChk END;
SYMBOL SwitchLabel: Sym: int;

RULE: SwitchLabel ::= 'case' Expression ':' COMPUTE
  SwitchLabel.Sym=
    CastPrimitive(
      Expression.Constant,
      Expression.Type,
      INCLUDING SwitchStatement.SwitchExpType);
END;

RULE: SwitchLabel ::= 'default' ':' COMPUTE
  SwitchLabel.Sym=MakeName("default");
END;
```

This macro is defined in definitions 17 and 18.
This macro is invoked in definition 14.

Instantiate required modules[19]:

```
$/Name/AlgScope.gnrc +instance=Case :inst
```

This macro is defined in definitions 7 and 19.
This macro is invoked in definition 2.

6.4.3 The break statement

The break statement[20]:

```
ATTR CannotBreak: int;
CLASS SYMBOL Breakable   COMPUTE SYNT.CannotBreak=0; END;
CLASS SYMBOL Unbreakable COMPUTE SYNT.CannotBreak=1; END;

TREE  SYMBOL LoopStatement    INHERITS Breakable   END;
TREE  SYMBOL SwitchStatement  INHERITS Breakable   END;
TREE  SYMBOL Goal             INHERITS Unbreakable END;
TREE  SYMBOL MethodBody       INHERITS Unbreakable END;
TREE  SYMBOL ClassInitializer INHERITS Unbreakable END;

RULE: Statement ::= 'break' ';' COMPUTE
  IF(INCLUDING (Breakable.CannotBreak,Unbreakable.CannotBreak),
    message(ERROR,"Plain break must occur in a loop or switch",0,COORDREF));
END;

TREE SYMBOL LabeledStatement COMPUTE SYNT.CannotBreak=0; END;

RULE: Statement ::= 'break' LabelIdUse ';' COMPUTE
  IF(INCLUDING (Breakable.CannotBreak,Unbreakable.CannotBreak,
                LabeledStatement.CannotBreak),
```

```
    message(
      ERROR,
      "Labeled break must occur in a loop, switch, or labeled statement",
      0,
      COORDREF));
END;
```

This macro is invoked in definition 14.

6.4.4 The continue statement

The continue statement[21]:

```
ATTR CannotContinue: int;
CLASS SYMBOL Continuable   COMPUTE SYNT.CannotContinue=0; END;
CLASS SYMBOL Uncontinuable COMPUTE
  SYNT.CannotContinue=1;
  SYNT.GotNotLoop=1;
END;

TREE SYMBOL LoopStatement    INHERITS Continuable   END;
TREE SYMBOL Goal             INHERITS Uncontinuable END;
TREE SYMBOL MethodBody       INHERITS Uncontinuable END;
TREE SYMBOL MethodBody       INHERITS Uncontinuable END;
TREE SYMBOL ClassInitializer INHERITS Uncontinuable END;

ATTR NotLoop: int;
TREE SYMBOL Statement             COMPUTE SYNT.NotLoop=1;      END;
RULE: Statement ::= LoopStatement COMPUTE Statement.NotLoop=0; END;
RULE: LabeledStatement ::= LabelIdDef ':' Statement COMPUTE
  LabeledStatement.GotNotLoop=
    ResetNotLoop(LabelIdDef.LblKey,Statement.NotLoop)
    <- INCLUDING (LabeledStatement.GotNotLoop,Uncontinuable.GotNotLoop);
END;

RULE: Statement ::= 'continue' ';' COMPUTE
  IF(INCLUDING (Continuable.CannotContinue,Uncontinuable.CannotContinue),
    message(ERROR,"Continue must occur in a loop or switch",0,COORDREF));
END;

RULE: Statement ::= 'continue' LabelIdUse ';' COMPUTE
  IF(INCLUDING (Continuable.CannotContinue,Uncontinuable.CannotContinue),
    message(ERROR,"No non-local jumps",0,COORDREF));
  IF(GetNotLoop(LabelIdUse.LblKey,1),
    message(
      ERROR,
      CatStrInd("Must label a loop: ",LabelIdUse.Sym),
      0,
      COORDREF))
    <- INCLUDING (LabeledStatement.GotNotLoop,Uncontinuable.GotNotLoop);
END;
```

This macro is invoked in definition 14.

Property definitions[22]:

```
NotLoop: int;
```

This macro is defined in definitions 10, 16, 22, 36, and 40.
This macro is invoked in definition 3.

6.4.5 The return statement

The return statement[23]:

```
RULE: Statement ::= 'return' ';' COMPUTE
  IF(NE(
      FinalType(INCLUDING (MethodBody.ReturnType,
                           ConstructorDeclaration.ReturnType,
                           ClassInitializer.ReturnType)),
      voidType),
    message(ERROR,"Must return a value here",0,COORDREF));
END;
```

This macro is invoked in definition 14.

6.4.6 The synchronized statement

The synchronized statement[24]:

```
RULE: Statement ::= 'synchronized' '(' Expression ')' Block COMPUTE
  IF(NOT(IsReferenceType(Expression.Type)),
    message(
      ERROR,
      "The expression must yield a reference type",
      0,
      COORDREF))
    <- INCLUDING Goal.GotAllConstants;
END;
```

This macro is invoked in definition 14.

6.4.7 The try statement

The try statement[25]:

```
RULE: CatchClause ::= 'catch' '(' FormalParameter ')' Block COMPUTE
  IF(NOT(IsCoercible(FinalType(FormalParameter.Type),throwableType)),
    message(ERROR,"Catch parameter must be throwable",0,COORDREF))
    <- INCLUDING Goal.GotAllConstants;
END;
```

This macro is invoked in definition 14.

6.5 Expressions

Expressions[26]:

This macro is invoked in definition 1.

6.5.1 Method invocation expressions

Method invocation expressions[27]:

```
RULE: Expression ::= MethodName '(' Arguments ')' COMPUTE
  IF(AND(NE(MethodName.Key,NoKey),BadOperator),
    message(ERROR,"Invalid method call",0,COORDREF));
END;

RULE: Expression ::= 'super' '.' MethodIdUse '(' Arguments ')' COMPUTE
  IF(AND(NE(MethodIdUse.Key,NoKey),BadOperator),
    message(ERROR,"Invalid method call",0,COORDREF));
END;

RULE: Expression ::= Expression '.' MethodIdUse '(' Arguments ')' COMPUTE
  IF(AND(NE(MethodIdUse.Key,NoKey),BadOperator),
    message(ERROR,"Invalid method call",0,COORDREF));
END;
```

This macro is invoked in definition 26.

6.5.2 Field access

Field access[28]:

```
TREE SYMBOL FieldIdUse INHERITS ChkTypedUseId END;
```

This macro is invoked in definition 26.

6.5.3 Array access

Array access[29]:

```
RULE: Expression ::= Expression '[' Expression ']' COMPUTE
  IF(BadOperator,
    message(ERROR,"Invalid array reference",0,COORDREF));
END;
```

This macro is invoked in definition 26.

6.5.4 Expression names

Expression names[30]:

```
TREE SYMBOL pExpressionName INHERITS ChkTypedUseId END;
```

This macro is invoked in definition 26.

6.5.5 Relational operators

Relational operators[31]:

```
RULE: Expression ::= Expression 'instanceof' Type COMPUTE
  IF(NOT(IsReferenceType(Expression[2].Type)),
    message(
      ERROR,
      "Left operand of instanceof must be a reference type",
      0,
      COORDREF))
    <- INCLUDING RootType.TypeIsSet;
  IF(NOT(IsReferenceType(FinalType(Type.Type))),
    message(
      ERROR,
      "Right operand of instanceof must be a reference type",
      0,
      COORDREF))
    <- INCLUDING RootType.TypeIsSet;
END;
```

This macro is invoked in definition 26.

6.5.6 Constant expressions

Constant expressions[32]:

```
ATTR Constant: int;

TREE SYMBOL Expression COMPUTE
  SYNT.Constant=NoStrIndex;
END;

RULE: Expression ::= 'false' COMPUTE
  Expression.Constant=MakeName("false");
END;

RULE: Expression ::= 'true' COMPUTE
  Expression.Constant=MakeName("true");
END;

RULE: Expression ::= CharacterLiteral COMPUTE
  Expression.Constant=CharacterLiteral;
END;

RULE: Expression ::= IntLiteral COMPUTE
  Expression.Constant=IntLiteral;
END;

RULE: Expression ::= LongLiteral COMPUTE
  Expression.Constant=LongLiteral;
END;
```

```
RULE: Expression ::= FloatLiteral COMPUTE
  Expression.Constant=FloatLiteral;
END;

RULE: Expression ::= DoubleLiteral COMPUTE
  Expression.Constant=DoubleLiteral;
END;

RULE: Expression ::= StringLiteral COMPUTE
  Expression.Constant=StringLiteral;
END;

RULE: Expression ::= 'null' COMPUTE
  Expression.Constant=MakeName("null");
END;
```

This macro is defined in definitions 32, 33, 37, 38, and 39.
This macro is invoked in definition 26.

Because a constant expression may depend on identifier values that are defined by other constant expressions, a LIDO iteration must be used to evaluate them during analysis. `Cluster.DoJinit` is the iteration attribute:

Constant expressions[33]:

```
RULE: Goal ::= Cluster COMPUTE
  Cluster.DoJinit=HaveNoJinit() <- Goal.GotAllOpers;
  Goal.GotAllConstants=
    UNTIL NoJinit <- CONSTITUENTS Initializer.GotConstants
    ITERATE Cluster.DoJinit=HaveNoJinit();
END;
```

This macro is defined in definitions 32, 33, 37, 38, and 39.
This macro is invoked in definition 26.

The general strategy is to set the global variable `NoJinit` to 1 at the beginning of each iteration, and then set it to 0 at any point during the iteration that an identifier obtains a value:

void HaveNoJinit(void)[34]:

```
/* Signal that no field or variable has been initialized
 *   On exit-
 *     NoJinit=1
 ***/
{ NoJinit = 1; }
```

This macro is invoked in definition 43.

void DidJinit(void)[35]:

```
/* Signal that a field or variable was initialized
 *   On exit-
 *     NoJinit=0
 ***/
{ NoJinit = 0; }
```

This macro is invoked in definition 43.

`HaveNoJinit` is invoked prior to the start of each iteration by the code of the LIDO iteration given above. `DidJinit` is invoked when the `Constant` property of an identifier is set for the first time:

Property definitions[36]:

```
Constant: int [Jinit];

void Jinit(DefTableKey key, int v)
{ if (key == NoKey || v == NoStrIndex) return;
  if (!ACCESS) DidJinit();
  VALUE = v;
}       "Context.h"
```

This macro is defined in definitions 10, 16, 22, 36, and 40.
This macro is invoked in definition 3.

The dependence stated in the `UNTIL` expression ensures that all initialization nodes are visited on each iteration:

Constant expressions[37]:

```
CHAIN IsFinal: int;

TREE SYMBOL TypeDeclaration  COMPUTE CHAINSTART HEAD.IsFinal=0; END;
TREE SYMBOL FieldDeclaration COMPUTE CHAINSTART HEAD.IsFinal=0; END;
RULE: Modifier ::= 'final'   COMPUTE        Modifier.IsFinal=1; END;

RULE: LocalVariableDeclaration ::= 'final' Type VariableDeclarators COMPUTE
  CHAINSTART Type.IsFinal=1;
END;

RULE: LocalVariableDeclaration ::=         Type VariableDeclarators COMPUTE
  CHAINSTART Type.IsFinal=0;
END;

TREE SYMBOL Initializer COMPUTE THIS.GotConstants="yes"; END;

RULE: FieldDeclarator ::= FieldDeclaratorId '=' Initializer COMPUTE
  Initializer.GotConstants=
    IF(FieldDeclaratorId.IsFinal,
      JinitConstant(
        FieldDeclaratorId CONSTITUENT FieldIdDef.Key,
        Initializer.Constant))
    <- INCLUDING Cluster.DoJinit;
END;

RULE: VariableDeclarator ::= VariableDeclaratorId '=' Initializer COMPUTE
  Initializer.GotConstants=
    IF(VariableDeclaratorId.IsFinal,
      JinitConstant(VariableDeclaratorId.Key,Initializer.Constant))
    <- INCLUDING Cluster.DoJinit;
```

```
END;

TREE SYMBOL VariableDeclaratorId: Key: DefTableKey;

RULE: VariableDeclaratorId ::= VariableDeclaratorId '[' ']' COMPUTE
  VariableDeclaratorId[1].Key=VariableDeclaratorId[2].Key;
END;

RULE: VariableDeclaratorId ::= VariableIdDef COMPUTE
  VariableDeclaratorId.Key=VariableIdDef.Key;
END;

RULE: Initializer ::= Expression COMPUTE
  Initializer.Constant=
    CastPrimitive(
      Expression.Constant,
      Expression.Type,
      INCLUDING (FieldDeclarators.Type,VariableDeclarators.Type));
END;

RULE: Initializer ::= '{' Initializers '}' COMPUTE
  Initializer.Constant=NoStrIndex;
END;

RULE: Expression ::= Name $pExpressionName COMPUTE
  Expression.Constant=
    GetConstant(pExpressionName.Key,NoStrIndex) <- INCLUDING Cluster.DoJinit;
END;
```

This macro is defined in definitions 32, 33, 37, 38, and 39.
This macro is invoked in definition 26.

A narrowing conversion of a signed integer to an integral type `T` simply discards all but the n lowest order bits, where n is the number of bits used to represent type `T`. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the sign of the resulting value to differ from the sign of the input value.

Constant expressions[38]:

```
RULE: Expression ::= '(' PrimitiveType ')' Expression COMPUTE
  Expression[1].Constant=
    CastPrimitive(
      Expression[2].Constant,
      Expression[2].Type,
      PrimitiveType.Type);
END;
```

This macro is defined in definitions 32, 33, 37, 38, and 39.
This macro is invoked in definition 26.

Constant expressions[39]:

```
RULE: Expression ::= Operator Expression COMPUTE
```

```
  Expression[1].Constant=
    APPLY(GetMonadicOp(Operator.Oper,StrBad1),Expression[2].Constant);
END;

RULE: Expression ::= Expression Operator Expression COMPUTE
  Expression[1].Constant=
    APPLY(
      GetDyadicOp(Operator.Oper,StrBad2),
      Expression[2].Constant,
      Expression[3].Constant);
END;

RULE: Expression ::= Expression '?' Expression ':' Expression COMPUTE
  Expression[1].Constant=
    IF(EQ(Expression[2].Constant,MakeName("true")),
      Expression[3].Constant,
      Expression[4].Constant);
END;
```

This macro is defined in definitions 32, 33, 37, 38, and 39.
This macro is invoked in definition 26.

Property definitions[40]:

```
MonadicOp: StrOp1;      "StrArith.h"
DyadicOp:  StrOp2;      "Math.h"

posOp -> MonadicOp={StrNop};
negOp -> MonadicOp={StrNeg};
invOp -> MonadicOp={StrNot};
addOp -> DyadicOp={StrAdd};
subOp -> DyadicOp={StrSub};
mulOp -> DyadicOp={StrMul};
divOp -> DyadicOp={StrDiv};
remOp -> DyadicOp={StrRem};
cmpleOp -> DyadicOp={StrLeq};
cmplsOp -> DyadicOp={StrLss};
cmpgeOp -> DyadicOp={StrGeq};
cmpgtOp -> DyadicOp={StrGtr};
cmpeqOp -> DyadicOp={StrEqu};
cmpneOp -> DyadicOp={StrNeq};
clseqOp -> DyadicOp={StrEqu};
clsneOp -> DyadicOp={StrNeq};
strprmOp -> DyadicOp={StrPrm};
prmstrOp -> DyadicOp={PrmStr};
strclsOp -> DyadicOp={StrStr};
clsstrOp -> DyadicOp={StrStr};
```

This macro is defined in definitions 10, 16, 22, 36, and 40.
This macro is invoked in definition 3.

6.6 Support code

Context.head[41]:

```
#include "Context.h"
```

This macro is attached to a product file.

Context.h[42]:

```
#ifndef CONTEXT_H
#define CONTEXT_H

#include "deftbl.h"
#include "eliproto.h"

extern int NoJinit;

extern void DidJinit ELI_ARG((void));
extern void HaveNoJinit ELI_ARG((void));
extern int NoAssign ELI_ARG((DefTableKey, DefTableKey, int));

#endif
```

This macro is attached to a product file.

Context.c[43]:

```
#include <stdio.h>
#include "csm.h"
#include "pdl_gen.h"
#include "Context.h"

int NoJinit;

void
#ifdef PROTO_OK
DidJinit(void)
#else
DidJinit)()
#endif
```
void DidJinit(void)[35]

```
void
#ifdef PROTO_OK
HaveNoJinit(void)
#else
HaveNoJinit()
#endif
```
void HaveNoJinit(void)[34]

```
int
#ifdef PROTO_OK
NoAssign(DefTableKey from, DefTableKey to, int v)
```

```
#else
NoAssign(from, to, v) DefTableKey from, to; int v;
#endif
```

int NoAssign(DefTableKey from, DefTableKey to, int v)[9]

This macro is attached to a product file.

Appendix A

Package Storage in the File System

As an extremely simple example, all the Java packages and source and binary code on a system might be stored in a single directory and its subdirectories. Each immediate subdirectory of this directory would represent a top-level package, that is, one whose fully qualified name consists of a single simple name. Subpackages of a package would be represented by subdirectories of the directory representing the package.

Each directory corresponding to a package might also contain `.java` files and `.class` files. Each of the `.java` files contains the source for a compilation unit that contains the definition of a class or interface whose binary compiled form is contained in the corresponding `.class` file.

A.1 Initialize the module

The standard JavaSoft Java Developer's Kit on UNIX stores all the Java packages and source and binary code in a set of directories and their subdirectories. This specification allows a user to provide the names of top-level directories via the `-I` option on the command line.

FileStorage.clp[1]:

```
PackageDirectories "-I" strings;
Source input;
```

This macro is attached to a product file.

The program also queries the `CLASSPATH` environment variable for additional directory names. Those directory names are appended to the `PackageDirectories` list. `CLASSPATH` must be read from the environment and the colon-separated directory names extracted:

void InitHostSystem(void)[2]:

```
{ char *dirs, *name;
  int SourceSym;

  dirs = getenv("CLASSPATH");
  if (dirs) {
    do {
      register char *rest;

      for (rest = dirs; *rest && *rest != ':'; rest++) ;

      { DefTableKey k = NewKey();
```

```
            CsmStrPtr = (char *)obstack_copy0(Csm_obstk, dirs, rest - dirs);
            ResetClpValue(k, MakeName(CsmStrPtr));
            PackageDirectories = AppElDefTableKeyList(PackageDirectories, k);
          }

          if (*rest == '\0') break;
          dirs = rest + 1;
        } while (*dirs);
      }

      if (!FilRootEnv) FilRootEnv = NewEnv();

      if (!PkgRootEnv) PkgRootEnv = NewEnv();
      if (!PtyRootEnv) PtyRootEnv = NewEnv();
      ImportPackage(
        "java.lang",
        DefineIdn(PkgRootEnv, MakeName("java.lang")));

      SourceSym = GetClpValue(Source, NoStrIndex);
      if (SourceSym == NoStrIndex) return;

      name = StringTable(SourceSym);
      if (*name == '/') SourceSym = MakeName(name);
      else {
        char cwd[MAX_PATH+1];

        if (!getcwd(cwd, MAX_PATH+1)) {
          (void)perror(cwd);
        }

        obstack_grow(Csm_obstk, cwd, strlen(cwd));
        obstack_1grow(Csm_obstk, '/');
        CsmStrPtr = (char *)obstack_copy0(Csm_obstk, name, strlen(name));
        SourceSym = MakeName(CsmStrPtr);
      }

      ResetIsDone(DefineIdn(FilRootEnv, SourceSym), 1);
    }
```

This macro is invoked in definition 9.

FileStorage.init[3]:

```
InitHostSystem();
```

This macro is attached to a product file.

A.2 char *DirectoryFor(char *pkg, int len)

`DirectoryFor` seeks the directory corresponding to the package in each of the directories specified by the given paths:

*char *DirectoryFor(char *pkg, int len)*[4]:

```
{ DefTableKeyList l;

  if (!pkg || *pkg == '\0') {
    char cwd[MAX_PATH+1];

    if (!getcwd(cwd, MAX_PATH+1)) {
      (void)perror(cwd);
      return;
    }

    return StringTable(MakeName(cwd));
  }

  for (l = PackageDirectories;
       l != NULLDefTableKeyList;
       l = TailDefTableKeyList(l)) {
    char *clpv;
    struct stat data;
    int i;

    clpv = StringTable(GetClpValue(HeadDefTableKeyList(l),0));
    obstack_grow(Csm_obstk, clpv, strlen(clpv));
    obstack_1grow(Csm_obstk,'/');

    for (i = 0; i < len; i++) {
      obstack_1grow(Csm_obstk, pkg[i] == '.' ? '/' : pkg[i]);
    }
    CsmStrPtr = (char *)obstack_copy0(Csm_obstk, "/", 1);

    if (!stat(CsmStrPtr, &data) && S_ISDIR(data.st_mode))
      return StringTable(MakeName(CsmStrPtr));

    obstack_free(Csm_obstk, CsmStrPtr);
  }

  return NULL;
}
```

This macro is invoked in definition 9.

A.3 Import a specific type

A single-type-import declaration makes that type available.

*void ImportType(char *typ)*[5]:

```
/* Define type names
 *   On entry-
 *     typ is the type to be made available
 *   If typ has a .java file then on exit-
 *     The .java file defining the type has been defined in FilRootEnv
 ***/
```

```
{ struct dirent *this_entry;
  char *filePart, *dir;
  struct stat status;

  if (typ == NULL || (filePart = strrchr(typ, '.')) == NULL) return;

  dir = DirectoryFor(typ, strlen(typ) - strlen(filePart++));
  if (dir == NULL) return;

  obstack_grow(Csm_obstk, dir, strlen(dir));
  obstack_grow(Csm_obstk, filePart, strlen(filePart));
  CsmStrPtr = obstack_copy0(Csm_obstk, ".java", 5);

  if (stat(CsmStrPtr, &status) || !S_ISREG(status.st_mode)) {
    (void)perror(CsmStrPtr);
    obstack_free(Csm_obstk, CsmStrPtr);
    return;
  }

  BindInScope(FilRootEnv, MakeName(CsmStrPtr));
}
```

This macro is invoked in definition 9.

A.4 Import files from a package

The scope of a type introduced by a class or interface declaration is the declarations of all class and interface declarations of all compilation units of the package in which it is declared. When a package declaration is encountered in a program, the compiler must make all class and interface declarations of that package available. Similarly, an import-on-demand declaration makes all of those types available.

*void ImportPackage(char *pkg, DefTableKey key)*[6]:

```
/* Import package files
 *   On entry-
 *     pkg is the name of the package whose files are to be imported
 *     key is the key under which the package's type environment
 *       is to be stored
 *     pkg has not yet been examined
 *   On exit-
 *     The package's type environment has been established and stored
 *     Its fully-qualified types have been defined in TypRootEnv
 *     The .java files in pkg have been defined in FilRootEnv
 ***/
{ DIR *package;
  char cwd[MAX_PATH+1], *dir;
  struct dirent *this_entry;
  int dirlen;
  Environment env;

  if((dir = DirectoryFor(pkg, strlen(pkg))) == NULL) return;
```

```
dirlen = strlen(dir);

if (!getcwd(cwd, MAX_PATH+1)) {
  (void)perror(cwd);
  return;
}

package = opendir(dir);
if (!package) {
  (void)perror(dir);
  return;
}

if (chdir(dir)) {
  (void)perror(dir);
  if (closedir(package)) (void)perror(dir);
  return;
}

env = NewScope(PtyRootEnv);
ResetPtyScope(key, env);
ResetTypScope(key, env);

while(this_entry = readdir(package)) {
  if ((strcmp(this_entry->d_name, ".")  != 0) &&
      (strcmp(this_entry->d_name, "..") != 0)) {
    struct stat status;

    if (stat(this_entry->d_name, &status)) {
      (void)perror(this_entry->d_name);
      break;
    }
    if (S_ISREG(status.st_mode)) {
    char *tail;

      if (tail = strchr(this_entry->d_name, '.')) {
        if (strcmp(tail, ".java") == 0) {
          Binding bind;
          int len = strlen(this_entry->d_name);

          obstack_grow(Csm_obstk, dir, dirlen);
          CsmStrPtr = obstack_copy0(Csm_obstk, this_entry->d_name, len);
          BindInScope(FilRootEnv, MakeName(CsmStrPtr));
        }
      }
    }
  }
}

if (chdir(cwd)) (void)perror(cwd);
if (closedir(package)) (void)perror(dir);
```

```
}
```

This macro is invoked in definition 9.

A.5 Advance to the next compilation unit

When the end of the current compilation unit is reached, `AnotherCompilationUnit` will be invoked to determine whether another compilation unit is available.

The Java file may be present but not readable, in which case `CompileType` must issue an error report and return without disturbing the source module:

int AnotherCompilationUnit(void)[7]:

```
{ int fd;
  Binding bind;
  char *name;

  for (bind = DefinitionsOf(FilRootEnv);
       GetIsDone(KeyOf(bind), 0);
       bind = NextDefinition(bind)) ;

  if (bind == NoBinding) return 0;

  ResetIsDone(KeyOf(bind), 1);

  name = StringTable(IdnOf(bind));

  if ((fd = open(name, 0)) == -1) {
    (void)perror(name);
    return 1;
  }

  (void)close(finlBuf());
  initBuf(name, fd);
  AddBreak(name, 1);
#ifdef MONITOR
  _dapto_source_changed(name, 1, LineNum, 1, 1, 1);
#endif
  return 1;
}
```

This macro is invoked in definition 9.

A.6 Generated Files

A type-`h` file defines the information exported by the module:

FileStorage.h[8]:

```
#ifndef FILESTORAGE_H
#define FILESTORAGE_H
```

```
#include "deftbl.h"

extern void ImportPackage ELI_ARG((char*, DefTableKey));
extern void ImportType ELI_ARG((char*));
extern int  AnotherCompilationUnit ELI_ARG((void));
extern void InitHostSystem ELI_ARG((void));

#endif
```

This macro is attached to a product file.

The implementation of the module is done in C:

FileStorage.c[9]:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>

#ifdef MONITOR
#include "clp_dapto.h"
#endif
#include "clp.h"
#include "err.h"
#include "csm.h"
#include "source.h"
#include "envmod.h"
#include "Strings.h"
#include "MakeName.h"
#include "pdl_gen.h"
#include "DefTableKeyList.h"
#include "PkgAlgScope.h"
#include "PtyAlgScope.h"
#include "FilAlgScope.h"
#include "FileStorage.h"

#define MAX_PATH 256

static char *
#if PROTO_OK
DirectoryFor(char *pkg, int len)
#else
DirectoryFor(pkg, len) char *pkg; int len;
#endif
```

*char *DirectoryFor(char *pkg, int len)*[4]

```
int
```

```
#if PROTO_OK
AnotherCompilationUnit(void)
#else
AnotherCompilationUnit()
#endif
```
int AnotherCompilationUnit(void)[7]

```
void
#if PROTO_OK
ImportType(char *typ)
#else
ImportType(typ) char *typ;
#endif
```
*void ImportType(char *typ)*[5]

```
void
#if PROTO_OK
ImportPackage(char *pkg, DefTableKey key)
#else
ImportPackage(pkg, key) char *pkg; DefTableKey key;
#endif
```
*void ImportPackage(char *pkg, DefTableKey key)*[6]

```
void
#if PROTO_OK
InitHostSystem(void)
#else
InitHostSystem()
#endif
```
void InitHostSystem(void)[2]

This macro is attached to a product file.

This module adds properties for its own use:

FileStorage.pdl[10]:

```
IsDone: int;
PtyScope: Environment;
```

This macro is attached to a product file.

FileStorage.specs[11]:

```
$/Tech/Strings.specs
$/Tech/MakeName.gnrc +instance=Identifier :inst
$/Name/AlgScope.gnrc +instance=Fil +referto=Fil :inst
$/Name/AlgScope.gnrc +instance=Pty +referto=Pty :inst
$/Name/AlgScope.gnrc +instance=Typ +referto=Typ :inst
```

This macro is attached to a product file.

Appendix B

Provide Readable Type Names

DebugName.specs[1]:

```
$/Tech/Strings.specs
```

This macro is attached to a product file.

DebugName.pdl[2]:

```
DebugName: CharPtr; "Strings.h"

byteType   -> DebugName={"byte"};
shortType  -> DebugName={"short"};
intType    -> DebugName={"int"};
longType   -> DebugName={"long"};
charType   -> DebugName={"char"};
boolType   -> DebugName={"boolean"};

floatType  -> DebugName={"float"};
doubleType -> DebugName={"double"};

voidType   -> DebugName={"void"};
objectType -> DebugName={"java.lang.Object"};
stringType -> DebugName={"java.lang.String"};
throwableType -> DebugName={"java.lang.Throwable"};
nullType   -> DebugName={"null"};

addOp -> DebugName={"addOp"};
andOp -> DebugName={"andOp"};
arrCondOp -> DebugName={"arrCondOp"};
arreqOp -> DebugName={"arreqOp"};
arrneOp -> DebugName={"arrneOp"};
arrstrOp -> DebugName={"arrstrOp"};
bogusOp -> DebugName={"bogusOp"};
candOp -> DebugName={"candOp"};
castNumOp -> DebugName={"castNumOp"};
clsCondOp -> DebugName={"clsCondOp"};
```

```
clseqOp -> DebugName={"clseqOp"};
clsneOp -> DebugName={"clsneOp"};
clsstrOp -> DebugName={"clsstrOp"};
cmpeqOp -> DebugName={"cmpeqOp"};
cmpgeOp -> DebugName={"cmpgeOp"};
cmpgtOp -> DebugName={"cmpgtOp"};
cmpleOp -> DebugName={"cmpleOp"};
cmplsOp -> DebugName={"cmplsOp"};
cmpneOp -> DebugName={"cmpneOp"};
complOp -> DebugName={"complOp"};
conjOp -> DebugName={"conjOp"};
corOp -> DebugName={"corOp"};
decrOp -> DebugName={"decrOp"};
disjOp -> DebugName={"disjOp"};
divOp -> DebugName={"divOp"};
exorOp -> DebugName={"exorOp"};
incrOp -> DebugName={"incrOp"};
invOp -> DebugName={"invOp"};
lshiftOp -> DebugName={"lshiftOp"};
mulOp -> DebugName={"mulOp"};
narrowOp -> DebugName={"narrowOp"};
negOp -> DebugName={"negOp"};
obj2arrOp -> DebugName={"obj2arrOp"};
objCondOp -> DebugName={"objCondOp"};
orOp -> DebugName={"orOp"};
posOp -> DebugName={"posOp"};
prmCondOp -> DebugName={"prmCondOp"};
prmstrOp -> DebugName={"prmstrOp"};
remOp -> DebugName={"remOp"};
rshiftOp -> DebugName={"rshiftOp"};
strarrOp -> DebugName={"strarrOp"};
strclsOp -> DebugName={"strclsOp"};
strprmOp -> DebugName={"strprmOp"};
subOp -> DebugName={"subOp"};
urshiftOp -> DebugName={"urshiftOp"};
```

This macro is attached to a product file.

DebugName.lido[3]:

```
CLASS SYMBOL PointerDeclarator COMPUTE
  ResetDebugName(THIS.Type,"pointer");
END;

CLASS SYMBOL ArrayDeclarator COMPUTE
  ResetDebugName(THIS.Type,"array");
END;

CLASS SYMBOL FunctionDeclarator COMPUTE
  ResetDebugName(THIS.Type,"function");
END;
```

This macro is attached to a product file.