An Analyzer for ANSI C

D. J. Banta
D. L. Carroll
S. A. Maurich
J. R. Roccatagliata
P. A. Shanahan
C. R. Turner
W. M. Waite

August 30, 2008

This document assumes familiarity with the basic principles of compiler construction and access to the Eli documentation. It is made up of a collection of FunnelWeb files, one per chapter.

The actual specification is embedded in the text as macros, sequentially numbered within each chapter. A macro is a relevant chunk of the specification, with a name describing its purpose. It may have references to other macros as components. Our intent is that the names of the referenced macros carry sufficient information to make the chunk itself understandable.

Attached to each macro definition is the set of macro numbers in which it is used, and attached to each use is the macro number of its definition. This allows the reader to easily find more information about a component if necessary. Some macro definitions are broken up to allow insertion of explanatory text. In this case, each section of the definition is given a distinct sequential number and all of them are attached both to each definition and to each use.

This specification, originally developed as a class project in the spring semester of 1994, implements ANSI/ISO 9899-1990. Within the document, ANSI/ISO 9899-1990 is referred to as "the standard".

The specification was tested using Eli 4.5. A complete description of Eli, including the current public-domain source code, can be found on the web[1].

[1] http://www.cs.colorado.edu/ eliuser/

Contents

Chapter 1

Phrase Structure

The *phrase structure* of a language defines the form of the input representation. Annex B of ANSI/ISO 9899-1990 summarizes the phrase structure of C, and this chapter captures most of that information (pre-processor definitions are omitted).

A scanner and parser that verify lexical and syntactic correctness of pre-processed C text can be generated from this chapter. It can also be used as one component of a larger specification from which a C compiler or special-purpose analyzer could be generated, or it could form the basis for a specification of an extension to C.

Section 1.1 specifies the way in which input characters are grouped into tokens and comments. This task is made more difficult than usual in C because certain basic symbols with the lexical structure of an identifier need to be classified as type names. The details of this process, which requires feedback from an analysis of the syntactic structure, are given in Section 1.3.

Section 1.2 specifies the way in which the syntactic structure is derived from the sequence of tokens.

1.1 Lexical structure

A type-`gla` file contains the declarative specifications of the character strings to be recognized in the input text.

Phrase.gla[1]:

```
Lexical elements[2]
  C_COMMENT
  $#  (auxEOL)
```

This macro is attached to a product file.

Eli provides a *canned description* for C comments that implements the definition in Section 6.1.9 of the standard.

The C pre-processor normally leaves lines beginning with `#` in its output text to define the correspondence between that output and the input text. Here we ignore those lines by invoking the token processor `auxEOL`, which skips past the end of the current line.

1.1.1 Tokens

Tokens are the basic symbols from which a C text is composed. Their composition is defined in several ways in this specification, as shown in Sections 1.1.2 through 1.1.9.

1.1.2 Keywords

Keywords are described by literal terminals in the grammar. Eli does not require an additional specification.

1.1.3 Identifiers

There is a well-known ambiguity in C, resulting from the fact that ordinary identifiers can be type specifiers. It is illustrated by the line:

```
A (*B);
```

If `A` has been defined as a type name, then the line is a declaration of a variable `B` to be of type "pointer to `A`". (The parentheses surrounding "`*B`" are ignored – see Section 6.5.4 of the standard.) If `A` has not been defined as a type name, then the line is a call of the function `A` with the single parameter `*B`. This ambiguity cannot be resolved by the grammar, since the two constructs can appear in the same contexts.

This specification resolves the ambiguity via an interaction between the scanner and the parser, as described in Section 1.3. When an identifier defined as a type name is used, the scanner reports it as the token `typedef_name` (Standard, Section 6.5.2).

The semantic analysis task can be simplified if identifiers in two other contexts (definition of a type name and definition of a `struct` or `union` member) can also be given distinct tokens. Since the scanner/parser interaction mechanism must be in place to resolve the type name ambiguity, this distinction can be added at very little cost.

Lexical elements[2]:

```
identifier:   C_IDENTIFIER [IdnOrType]
typedef_name: C_IDENTIFIER
typedef_def:  C_IDENTIFIER
member_def:   C_IDENTIFIER
```

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

Note that all of these tokens have exactly the same lexical structure. The ambiguity resolution rules of Eli guarantee that the token processor `IdnOrType` will be invoked after each character sequence satisfying this definition has been recognized. `IdnOrType` will then classify the sequence. Details can be found in Section 1.3

1.1.4 Constants

The classification of constants is carried out by a syntactic rule:

Syntactic elements[3]:

```
constant:
  floating_constant /
  integer_constant /
  character_constant .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

A `constant` is one possible form of `primary_expression` (that is the only context in which `constant` appears in the grammar), and an `identifier` is another. Because an enumeration constant is an `identifier`, there would be an ambiguity in the grammar if `enumeration_constant` appeared in this definition of `constant` (as it does in the standard): Should the `identifier` be reduced directly to a `primary_expression`, or first to an `enumeration_constant`?

Information about the the declaration of an identifier is needed to distinguish an enumeration constant from a variable identifier in the context represented by `constant`, and this information is not available to the parser. Since the distinction is irrelevant in determining the structure of the tree anyway, it can simply be ignored.

Eli provides canned descriptions for C constants that implement the definitions found in Section 6.1.3.1, 6.1.3.2, and 6.1.3.4 of the standard:

Lexical elements[4]:

```
floating_constant:  C_FLOAT          [mkidn]
integer_constant:   C_INT_DENOTATION [mkidn]
character_constant: C_CHAR_CONSTANT  [mkidn]
```

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

This specification uses `mkidn` to guarantee that there is only one copy of any constant in the string table.

1.1.5 String literals

C string literals are also defined by an Eli canned description, and made unique:

Lexical elements[5]:

```
string_literal:  C_STRING_LIT [mkidn]
```

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

1.1.6 Operators

Operators are represented by literal symbols in the grammar, so no additional lexical specification is necessary.

1.1.7 Punctuators

Punctuators are represented by literal symbols in the grammar, so no additional lexical specification is necessary.

1.1.8 Header names

Preprocessing is not defined in this chapter.

1.1.9 Preprocessing numbers

Preprocessing is not defined in this chapter.

1.2 Syntactic structure

A type-con file contains the context-free grammar describing how the structure of a program is determined from the sequence of tokens.

Phrase.con[6]:

Syntactic elements[3]

This macro is attached to a product file.

In the syntactic notation used here, syntactic categories (nonterminals) are indicated by words in this type, and literal words and character set members (terminals) are enclosed in apostrophes ' '. The underscore character _ is used instead of the standard's hyphen - within symbols. A colon (:) following a nonterminal introduces its definition. Alternative definitions are separated by slashes (/), and the definition is terminated by a period.

CHANGEME: There should be a description of the BNF extensions here, as well as a brief discussion of the need to use them in order to implement certain lists.

An optional symbol is indicated by adding _opt to the name.

Optional symbols are defined implicitly in the standard, but explicit definitions are required in order to generate a parser:

Syntactic elements[7]:

```
type_qualifier_list_opt:      / type_qualifier_list .
constant_exp_opt:             / constant_expression .
expression_opt1:              / expression .
expression_opt2:              / expression .
argument_expression_list_opt: / argument_expression_list .
init_declarator_list_opt:     / init_declarator_list .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

1.2.1 Expressions

Syntactic elements[8]:

```
primary_expression:
  identifier /
  constant /
  StringSeq /
  '(' expression ')' .

StringSeq:
  string_literal /
  StringSeq string_literal .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Section 6.1.4 of the standard states that any sequence of adjacent character string literal tokens are concatenated into a single sequence when translated. We use the `StringSeq` phrase here to formalize that specification.

Syntactic elements[9]:

```
postfix_expression:
  primary_expression /
  postfix_expression '[' expression ']' /
  postfix_expression '(' argument_expression_list_opt ')' /
  postfix_expression '.' identifier /
  postfix_expression '->' identifier /
  postfix_expression '++' /
  postfix_expression '--' .

argument_expression_list:
  assignment_expression /
  argument_expression_list ',' assignment_expression.

unary_expression:
  postfix_expression /
  '++' unary_expression /
  '--' unary_expression /
  '&' cast_expression /
  unary_operator cast_expression /
  'sizeof' unary_expression /
  'sizeof' '(' type_name ')'.

unary_operator:
  '*' / '+' / '-' / '~'  / '!' .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Section 6.2.2.1 of the standard says that when the unary `&` operator is applied to an lvalue, the result is the address of the object designated by the lvalue. If any other unary operator is applied to an lvalue, that lvalue is converted to the value stored in the designated object. Thus the semantics of the unary `&` operator differ significantly from those of the other unary operators. Semantic analysis is simplified if the two contexts are distinguished.

Syntactic elements[10]:

```
cast_expression:
  unary_expression /
  '(' type_name ')' cast_expression.

multiplicative_expression:
  cast_expression /
  multiplicative_expression '*' cast_expression /
  multiplicative_expression '/' cast_expression /
  multiplicative_expression '%' cast_expression .
```

```
additive_expression:
  multiplicative_expression /
  additive_expression '+' multiplicative_expression /
  additive_expression '-' multiplicative_expression .

shift_expression:
  additive_expression /
  shift_expression '<<' additive_expression /
  shift_expression '>>' additive_expression .

relational_expression:
  shift_expression /
  relational_expression '<'  shift_expression /
  relational_expression '>'  shift_expression /
  relational_expression '<=' shift_expression /
  relational_expression '>=' shift_expression .

equality_expression:
  relational_expression /
  equality_expression '==' relational_expression /
  equality_expression '!=' relational_expression .

AND_expression:
  equality_expression /
  AND_expression '&' equality_expression.
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27. This macro is invoked in definition 6.

Syntactic elements[11]:

```
exclusive_OR_expression:
  AND_expression /
  exclusive_OR_expression '^' AND_expression.

inclusive_OR_expression:
  exclusive_OR_expression /
  inclusive_OR_expression '|' exclusive_OR_expression.

logical_AND_expression:
  inclusive_OR_expression /
  logical_AND_expression '&&' inclusive_OR_expression.

logical_OR_expression:
  logical_AND_expression /
  logical_OR_expression '||' logical_AND_expression.

conditional_expression:
  logical_OR_expression /
  logical_OR_expression '?' expression ':' conditional_expression .
```

```
assignment_expression:
  conditional_expression /
  unary_expression assignment_operator assignment_expression .

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

expression:
  assignment_expression /
  expression ',' assignment_expression .

constant_expression:
  conditional_expression .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

1.2.2 Declarations

A `declaration` specifies the interpretation and attributes of a set of identifiers. The `declaration_specifiers` phrase consists of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities denoted in the `init_declarator_list_opt`.

Syntactic elements[12]:

```
declaration:
  declaration_specifiers init_declarator_list_opt &'Reset state[31]' ';' .

init_declarator_list:
  init_declarator /
  init_declarator_list ',' init_declarator .

init_declarator:
  declarator &'Deferred binding[46] End parameters[53]' /
  declarator &'Deferred binding[46] End parameters[53]' '=' initializer .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Specifiers appearing in the `declaration_specifiers` phrase must be drawn from one of the three sets (storage class specifiers, type specifiers, type qualifiers) defined in Section 6.5 of the standard.

Five of the type specifiers can only appear in lists of one element; the other seven can be used to form lists of one or more elements. It is important to distinguish the two kinds of type specifiers syntactically in order to parse a sequence like "`int j;`" properly when `j` has been declared in an outer scope to be a `typedef_name`: If the parser knows that a `typedef_name` cannot follow the `type_specifier int`, then it will be forced to re-classify `j` as an identifier. If this information is not available to the parser, then it will detect a syntax error after accepting `j` as a `type_specifier` and not finding an identifier. At that point, however, it is too late to re-classify `j`.

The `type_specifier`s that must appear singly are therefore classified as `type_specifier_1`s, and those that may appear in groups are classified as `type_specifier_2`s.

The phrase `declaration_specifiers` is then defined to enforce the constraint syntactically. All lists must be left-recursive here in order to detect the error at the proper symbol:

Syntactic elements[13]:

```
declaration_specifiers:
  ds0 / ds1 / ds2.

  ds0:  /* List without type specifiers */
    storage_class_specifier / ds0 storage_class_specifier /
    type_qualifier / ds0 type_qualifier .

  ds1:  /* List with a single type_specifier_1 */
    type_specifier_1 / ds0 type_specifier_1 /
    ds1 storage_class_specifier / ds1 type_qualifier .

  ds2:  /* List with one or more type_specifier_2's */
    type_specifier_2 / ds0 type_specifier_2 /
    ds2 type_specifier_2 / ds2 storage_class_specifier / ds2 type_qualifier .

storage_class_specifier:
  'typedef' &'Accept a typedef specifier'[32]' /
  'extern' /
  'static' /
  'auto' /
  'register' .

type_specifier_1:
  'void' /
  'float' /
  struct_or_union_specifier /
  enum_specifier /
  typedef_name .

type_specifier_2:
  'char' /
  'short' /
  'int' /
  'long' /
  'double' /
  'signed' /
  'unsigned' .

struct_or_union_specifier:
  struct_or_union identifier
    &'Nest'[34]('member_def','0')' '{' struct_declaration_list &'Restore'[36]' '}' /
  struct_or_union
    &'Nest'[34]('member_def','0')' '{' struct_declaration_list &'Restore'[36]' '}' /
  struct_or_union identifier $';' .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

The modifier `$';'` in the last line above prevents a reduction to `struct_or_union_specifier` if the lookahead symbol is `;`. A modifier is only effective, however, if there is a conflict. There is no conflict in this

grammar involving that alternative: `struct_or_union identifier` may be followed by either a `declarator` or a `;` and the reduction to a `struct_or_union_specifier` is valid.

Section 6.5.2.3 of the standard gives the two cases quite different semantics, however. If `struct_or_union identifier` is followed by a `;` then the `identifier` is a structure or union tag that specifies a new type distinct from any type with the same tag in an enclosing scope (if any). Otherwise it will specify a new type only if there is no structure or union with that tag visible at that point.

When constructing an abstract syntax tree, it is useful to distinguish these contexts by first reducing the `identifier` to a separate nonterminal and then reducing the result. If this is done, and the given modifier is not present, then a conflict will be introduced into the grammar. The modifier is therefore inserted here in order to allow construction of an appropriate abstract syntax tree without the need to alter this specification.

Syntactic elements[14]:

```
struct_or_union:
  'struct' /
  'union' .

struct_declaration_list: struct_declaration+ .

struct_declaration:
  specifier_qualifier_list struct_declarator_list ';'.
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

A `specifier_qualifier_list` phrase differs from a `declaration_specifiers` phrase only in the fact that storage class specifiers are not allowed. The same problem with type specifiers must be solved, and the same mechanism is used.

Syntactic elements[15]:

```
specifier_qualifier_list:
  sq0 / sq1 / sq2 .

  sq0:  /* List without type specifiers */
    type_qualifier / sq0 type_qualifier .

  sq1:  /* List with a single type_specifier_1 */
    type_specifier_1 / sq0 type_specifier_1 / sq1 type_qualifier .

  sq2:  /* List with one or more type_specifier_2's */
    type_specifier_2 / sq0 type_specifier_2 /
    sq2 type_specifier_2 / sq2 type_qualifier .

struct_declarator_list: struct_declarator // ',' .

struct_declarator:
  member_declarator                          /
  member_declarator ':' constant_expression /
                    ':' constant_expression .
```

```
member_declarator:
  member_pointer_declarator /
  member_direct_declarator .

member_direct_declarator:
  MemberIdDef $')'          /
  '(' member_declarator ')' /
  member_array_declarator   /
  member_function_declarator .

member_pointer_declarator:
  '*' type_qualifier_list_opt member_declarator .

member_array_declarator:
  member_direct_declarator '[' constant_exp_opt ']' .

member_function_declarator:
  member_direct_declarator '(' empty_parameter_type_list                ')' /
  member_direct_declarator
    '(' &'Nest[34](`identifier',`0')' parameter_type_list &'Restore[36]' ')' .

MemberIdDef:
  member_def &'Bind[47]' /
  '(' MemberIdDef ')' .

enum_specifier:
  'enum' identifier
    &'Nest[34](`identifier',`1')' '{' enumerator_list &'Restore[36]' '}' /
  'enum'
    &'Nest[34](`identifier',`1')' '{' enumerator_list &'Restore[36]' '}' /
  'enum' identifier .

enumerator_list:
  enumerator /
  enumerator_list ',' enumerator .

enumerator:
  enumeration_constant /
  enumeration_constant '=' constant_expression.

enumeration_constant:
  identifier &'Bind[47]' .

type_qualifier:
  'const' /
  'volatile'.
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Declarators

Section 6.5.4 of the standard devotes separate subsections to pointer declarators, array declarators, and function declarators but does not distinguish these constructs syntactically. It is useful for semantic analysis to separate the forms of declarator that actually declare types from those that simply place identifiers.

The standard uses an optional `pointer`, and defines the `pointer` itself recursively. That definition not only leads to a grammar conflict, but the structure it imposes on the text does not correspond to the structure required by Section 6.5.4.1 of the standard.

Syntactic elements[16]:

```
declarator:
  pointer_declarator /
  direct_declarator .

direct_declarator:
  TypeIdDef   $'(' $')' &'Parameters?'[52]('0')' /
  IdDef       $'(' $')' &'Parameters?'[52]('0')' /
  '(' declarator ')' /
  array_declarator /
  function_declarator .

pointer_declarator:
  '*' type_qualifier_list_opt declarator .

type_qualifier_list:
  type_qualifier /
  type_qualifier_list type_qualifier .

array_declarator:
  direct_declarator '[' constant_exp_opt ']' .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

The block scope of a function definition begins at the beginning of the parameter list. Thus a (possible) parameter list of a function must be distinguished from the function prototypes that may form part of a complex declarator. The type rules in Section 6.5.4 of the standard show that the parameter list of a function definition is always the first modifier following the declarator's identifier. That identifier may, however, itself be enclosed in parentheses to arbitrary depth. The nonterminal `declarator_name` represents such an identifier. It can be reduced to a `direct_declarator` unless it is followed by a parameter list; in that case the following parameter list (represented by `parameters` is possibly the parameter list of the function definition.

Syntactic elements[17]:

```
function_declarator:
  direct_declarator '('        empty_parameter_type_list                ')' /
  direct_declarator
    '(' &'Nest'[34]('identifier','0')' parameter_type_list &'Restore'[36]' ')' /
  TypeIdDef         '(' &'Parameters?'[52]('1')' parameters              ')' /
```

```
  IdDef             '(' &'Parameters?'[52]('1')' parameters              ')' .

parameter_type_list:
  parameter_list /
  parameter_list ',' '...' .

parameter_list:
  parameter_declaration /
  parameter_list ',' parameter_declaration.

parameter_declaration:
  declaration_specifiers declarator &'Deferred binding'[46] 'End parameters'[53]' /
  declaration_specifiers abstract_declarator /
  declaration_specifiers  .

parameters:
  parameter_list_1 /
  parameter_list_1 ',' '...' /
  [ identifier &'Bind'[47]' // ',' ] .

parameter_list_1:
  parameter_declaration /
  parameter_list_1 ',' parameter_declaration.

TypeIdDef:
  typedef_def &'Defer the binding'[45]' /
  '(' TypeIdDef ')' .

IdDef:
  identifier &'Defer the binding'[45]' /
  '(' IdDef ')' .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Type names

The standard uses an optional `pointer` and an optional `direct_abs_declarator`. These options must be made explicit here in order to avoid a grammar conflict.

Syntactic elements[18]:

```
type_name:
  specifier_qualifier_list abstract_declarator /
  specifier_qualifier_list .

abstract_declarator:
  pointer_abstract_declarator /
  direct_abstract_declarator .

pointer_abstract_declarator:
```

```
  '*' type_qualifier_list_opt abstract_declarator .

direct_abstract_declarator:
  '(' abstract_declarator ')' /
  array_abstract_declarator /
  function_abstract_declarator .

array_abstract_declarator:
  direct_abstract_declarator '[' constant_exp_opt ']' /
  '[' constant_exp_opt ']' .

function_abstract_declarator:
  direct_abstract_declarator '(' empty_parameter_type_list ')' /
  direct_abstract_declarator
    '(' &'Nest[34](`identifier',`0')' parameter_type_list &'Restore[36]' ')' /
  '(' empty_parameter_type_list ')' /
  '(' &'Nest[34](`identifier',`0')' parameter_type_list &'Restore[36]' ')' .

empty_parameter_type_list: .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Initialization

The row initializer should be an abstract syntax tree node with an arbitrary number of children. Eli cannot construct such a list if the list has itself as a component. Therefore another symbol must be introduced to describe the top-level `initializer_list`.

Syntactic elements[19]:

```
initializer:
  assignment_expression /
  Row_initializer .

Row_initializer:
  '{' initializer_list '}' /
  '{' initializer_list ',' Empty_initializer '}'.

initializer_list:
  initializer /
  initializer_list ',' initializer.

Empty_initializer: .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

1.2.3 Statements

Syntactic elements[20]:

```
statement:
  labeled_statement /
  compound_statement /
  expression_statement /
  selection_statement /
  switch_statement /
  iteration_statement /
  jump_statement.
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

The left context of the `identifier` in a `labeled_statement` is identical to that of a `typedef_name` in a declaration. Therefore the parser cannot provide any information that would allow the lexical analyzer to distinguish the label from a named type.

In the absence of parser information, the lexical analyzer will classify an identifier as an `identifier` or `typedef_name` depending on the context. Thus the parser must accept either of these terminals as a valid label.

Syntactic elements[21]:

```
labeled_statement:
  identifier ':' statement /
  typedef_name ':' statement /
  'case' constant_expression ':' statement /
  'default' ':' statement.

compound_statement:
  &'New region'[49]' &'Nest'[34](`identifier`,`1`)' '{' body &'Restore'[36]' &'End region'[50]' '}' .

decls: declaration / decls declaration .
stmts: statement / stmts statement .
body: / decls / stmts / decls stmts .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Section 6.6.3 of the standard defines the expression and null statements in one syntax rule, using the `expression_opt` nonterminal. This implies that a null statement is really an expression statement in which the expression happens to be empty. Semantically, however, the null statement "performs no operation". Two alternatives express this situation more clearly:

Syntactic elements[22]:

```
expression_statement:
  expression ';' /
             ';' .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

There is a grammar conflict here between the first and second alternatives. This conflict is resolved by the modification `$'else'`, which prevents reduction of the first alternative if the lookahead symbol is `else`.

Syntactic elements[23]:

```
selection_statement:
  'if' '(' expression ')' statement $'else' /
  'if' '(' expression ')' statement 'else' statement.

switch_statement:
  'switch' '(' expression ')' statement.
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

According to Section 6.6.5.3 of the standard, the effects of omitting the initializer and the step of the `for` statement are the same; omitting the condition has a different effect. Since these effects must be distinguished in the abstract syntax tree, the grammar uses different symbols.

Syntactic elements[24]:

```
iteration_statement:
  'while' '(' expression ')' statement /
  'do' statement 'while' '(' expression ')' ';' /
  'for' '(' expression_opt1 ';' expression_opt2 ';' expression_opt1 ')'
    statement.
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

Section 6.6.6.2 of the standard defines the two versions of the return statement in one syntax rule, using the `expression_opt` nonterminal. These two versions have different constraints, however. Two alternatives express this situation more clearly:

Syntactic elements[25]:

```
jump_statement:
  'goto' identifier   ';' /
  'continue'          ';' /
  'break'             ';' /
  'return'            ';' /
  'return' expression ';' .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

1.2.4 External Definitions

Syntactic elements[26]:

```
program: external_declaration* .

external_declaration:
  function_definition /
  declaration .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

The standard uses an optional `declaration_specifiers`. This option must be made explicit here in order to avoid a grammar conflict.

Syntactic elements[27]:

```
function_definition:
  empty_declaration_specifiers
    declarator &'Deferred binding[46]' declaration_list function_body /
  declaration_specifiers
    declarator &'Deferred binding[46]' declaration_list function_body .

empty_declaration_specifiers: .

declaration_list: par_declaration* .

par_declaration: declaration_specifiers par_id_decls ';' .

par_id_decls: declarator // ',' .

function_body:
  '{' body &'End parameters[53]' '}' .
```

This macro is defined in definitions 3, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.
This macro is invoked in definition 6.

1.3 Identifier classification

In order to resolve the type name ambiguity, the scanner needs to distinguish ordinary identifiers that are defined as type names from those that are not. Two additional identifier classes can be used to embody important semantic properties:

- `typedef_name`: An applied occurrence of an ordinary identifier defined as a type name. This category of identifier is necessary to resolve an ambiguity.
- `typedef_def`: A defining occurrence of an ordinary identifier defined as a type name. This category of identifier is useful because the actions needed to define an identifier representing a type are different from those needed to define an identifier representing a typed object.
- `member_def`: A defining occurrence of a member identifier. This category of identifier is useful because a member identifier belongs to a different name space than an ordinary identifier. Applied occurrences of member identifiers occur in contexts that are syntactically distinct, but there is no such distinction for defining occurrences. This classification provides that distinction.
- `identifier`: Identifier occurrences whose processing requirements can be determined from their gramatical construct.

1.3.1 Reporting the classification

The specification given in Section 1.1.3 results in a scanner that invokes the token processor `IdnOrType` when it recognizes a character sequence obeying the definition of a C identifier. A token processor is responsible for reporting the syntactic classification of the token and its internal representation (if any) to the parser.

Invoking the token processor `mkidn` establishes the internal representation of the identifier. Here are the steps for classifying it:

1. If the identifier is bound to `typedef_name` in the current context, classify it as a `typedef_name`.
2. If the identifier lies within a `struct_declaration_list`, but not within the parameter list of a function declarator, classify it as a `member_def`.
3. If the identifier lies within an `init_declarator_list` in a declaration having a `typedef` storage class specifier, but not within the parameter list of a function declarator, classify it as a `typedef_def`.
4. Otherwise classify it as an `identifier`.

After classifying the identifier, `IdnOrType` saves some information for later binding:

*IdnOrType(char *start, int length, int *syncode, int *rep)*[28]:

```
/* Obtain the internal representation of an identifier
 *   On entry-
 *     start points to the character string for the identifier
 *     length=length of the character string for the identifier
 *     syncode points to a location containing the initial terminal code
 *     Context.SynCode in [member_def, typedef_def, identifier]
 *   On exit-
 *     syncode has been set to the terminal code
 *     rep has been set to the internal coding
 *     Context.Symbol=string table index of the token
 ***/
{
  mkidn(start, length, syncode, rep);

  *syncode = GetSynCode(KeyInEnv(CurrentEnv, *rep), Context.SynCode);
  if (*syncode == identifier && Context.SynCode != identifier)
    *syncode = Context.SynCode;

  Instance information for binding[41]
}
```

This macro is invoked in definition 55.

`CurrentEnv`, the set of bindings currently valid, is described in Section 1.3.3.

`IdnOrType` may deliver a classification that is not allowed by the grammar. In that case, the generated parser will call `Reparatur` in an attempt to get the token reclassified. Two reclassifications are possible:

*Reparatur(POSITION *coord, int *syncode, int *rep)*[29]:

```
/* Repair a syntax error by changing the lookahead token
 *   On entry-
```

```
 *     coord points to the coordinates of the lookahead token
 *     syncode points to the classification of the lookahead token
 *     rep points to the representation of the lookahead token
 *   If the lookahead token has been changed then on exit-
 *     Reparatur=1
 *     coord, syncode and rep reflect the change
 *   Else on exit-
 *     Reparatur=0
 *     coord, syncode and rep are unchanged
 ***/
{
  if (*syncode == typedef_name && Context.SynCode == member_def) {
    *syncode = member_def;
    return 1;
  }
  if (*syncode != identifier) { *syncode = identifier; return 1; }
  return 0;
}
```

This macro is invoked in definition 55.

1.3.2 Tracking current context

The following information is sufficient to track the current context for the purposes of classifying identifier occurrences:

State variable declarations[30]:

```
typedef struct {
  int SynCode;   /* Identifier classification on the basis of context */
  int BindingOK; /* 1 if identifiers should be bound in this context */
} IdProperties;

extern IdProperties Context;
```

This macro is defined in definitions 30, 37, and 39.
This macro is invoked in definition 56.

It is not possible to set the value of `Context` before each declaration, because that leads to a conflict in the grammar. Therefore the value must be reset at the beginning of the compilation and the end of each declaration:

Reset state[31]:

```
Context.SynCode = identifier; Context.BindingOK = 1;
```

This macro is invoked in definitions 12 and 59.

The classification is set to `typedef_def` when the storage class specifier `typedef` is accepted by the parser:

Accept a typedef specifier[32]:

```
Context.SynCode = typedef_def;
```

This macro is invoked in definition 13.

A `struct_declaration` may be nested within a normal declaration or another structure or union. Since the possible nesting depth is arbitrary, a stack is needed:

Instantiate a stack module for state values[33]:

```
$/Adt/Stack.gnrc +instance=IdState +referto=IdProperties :inst
```

This macro is invoked in definition 57.

The classification set to either `member_def` (for a `struct` or `union` declaration) or to `identifier` (for any other declaration). In addition, the binding permission should be set for an object or function declaration, but not for function prototypes.

Nest[34](◇2):

```
BeginDeclaration(◇1,◇2);
```

This macro is invoked in definitions 13, 15, 17, 18, 21, and 54.

BeginDeclaration(int c, int b)[35]:

```
{
  IdStateStackPush(Context);
  Context.SynCode = c; Context.BindingOK = b;
}
```

This macro is invoked in definition 55.

Restore[36]:

```
Context = IdStateStackTop; IdStateStackPop;
```

This macro is invoked in definitions 13, 15, 17, 18, 21, and 54.

1.3.3 Binding identifier occurrences

`IdnOrType` saves information about the identifier it classifies. If this is a defining occurrence of an ordinary identifier, and binding is permitted, then the given identifier should be bound in the current scope with the appropriate classification.

State variable declarations[37]:

```
typedef struct {
  int BindingOK;   /* 1 if binding is allowed */
  int Symbol;      /* Identifier to be bound */
  Environment Env; /* Environment in which to bind */
  int SynCode;     /* Identifier classification */
} BindData;

extern BindData CurrentId;
```

This macro is defined in definitions 30, 37, and 39.
This macro is invoked in definition 56.

Scope rules are embodied in the concept of an *environment*, which is a set of bindings for identifiers. The Eli library's `envmod` module implements a countour-model environment suitable for classifying C identifiers:

Instantiate an environment module for identifier analysis[38]:

```
$/Name/envmod.specs
```

This macro is invoked in definition 57.

State variable declarations[39]:

```
extern Environment CurrentEnv;
```

This macro is defined in definitions 30, 37, and 39.
This macro is invoked in definition 56.

Initialize the set of bindings[40]:

```
CurrentEnv = NewEnv();
```

This macro is invoked in definition 59.

`IdnOrType` captures the current state, adds the specific identifier, and determines the appropriate classification:

Instance information for binding[41]:

```
CurrentId.BindingOK = Context.BindingOK;
CurrentId.Symbol    = *rep;
CurrentId.Env       = CurrentEnv;
CurrentId.SynCode   = Context.SynCode == typedef_def ? typedef_name
                                                     : identifier;
```

This macro is invoked in definition 28.

The environment module binds the identifier to a definition table key; the classification must be stored as a property of that key:

Phrase.pdl[42]:

```
SynCode: int;
```

This macro is attached to a product file.

Bind(void)[43]:

```
{
  if (CurrentId.BindingOK)
    ResetSynCode(DefineIdn(CurrentId.Env, CurrentId.Symbol), CurrentId.SynCode);
}
```

This macro is invoked in definition 55.

In some constructs the state must be saved and the actual binding made when the end of that construct is reached; in other cases the binding must be done immediately. Deferred bindings are nested, so they can be stored on a stack:

Instantiate a stack module for deferred bindings[44]:

```
$/Adt/Stack.gnrc +instance=Bind +referto=BindData :inst
```

This macro is invoked in definition 57.

Defer the binding[45]:

```
BindStackPush(CurrentId);
```

This macro is invoked in definition 17.

Deferred binding[46]:

```
CurrentId = BindStackTop; BindStackPop; Bind();
```

This macro is invoked in definitions 12, 17, and 27.

Bind[47]:

```
Bind();
```

This macro is invoked in definitions 15 and 17.

1.3.4 Nested environments

Environments can be nested, in which case the binding in the innermost environment hides those in outer environments. This means that a stack can be used to store the `Environment` values:

Instantiate a stack module for environment values[48]:

```
$/Adt/Stack.gnrc +instance=Region +referto=Environment :inst
```

This macro is invoked in definition 57.

New region[49]:

```
RegionStackPush(CurrentEnv); CurrentEnv = NewScope(CurrentEnv);
```

This macro is invoked in definitions 21 and 54.

End region[50]:

```
CurrentEnv = RegionStackTop; RegionStackPop;
```

This macro is invoked in definitions 21 and 54.

1.3.5 Procedure parameters

According to Section 6.1.2.1 of the standard, an identifier declared in the parameter list of a function of a function has block scope, which terminates at the } closing the function definition. That means we need to create an environment at the beginning of the parameter list and discard it at the end of the function definition.

The problem is that when the beginning of the parameter list is encountered there is no way to tell whether the declarator is part of a function definition or not.

If the declarator is part of a nested declaration, then its parameter list is a function prototype and the environment can be discarded at the end of the parameter list.

If the declarator is at the outermost level of an external declaration, then its parameter list *may* be the parameter list of a function and the environment cannot be discarded at the end of the parameter list. It must be discarded at the end of the declaration.

A particular external declaration may or may not have a declarator specifying a parameter list. The current environment should be discarded at the end of an external declaration if and only if the declaration actually has a parameter list.

The necessary information can be provided by a variable that distinguishes four possible states:

State variable definitions[51]:

```
typedef enum {
  External,
  NoParameters,
  HasParameters,
  IsNested
} DeclarationStateValues;

static int DeclarationState = External;
```

This macro is defined in definitions 51 and 54.
This macro is invoked in definition 55.

Arbitrary nesting is handled without a stack by incrementing the current value of `DeclarationState` by `IsNested`: A value greater than or equal to `IsNested` indicates the nested state, but the state at the top level is also preserved. This is the reason that `DeclarationState` cannot be declared as a variable of type `DeclarationStateValues`; with arbitrary nesting, `DeclarationState` takes on values that are not valid `DeclarationStateValues`.

Parameters?[52](◇1):

```
BeginParameters(◇1);
```

This macro is invoked in definitions 16 and 17.

End parameters[53]:

```
EndDeclaration();
```

This macro is invoked in definitions 12, 17, and 27.

State variable definitions[54]:

```
void
BeginParameters(int p)
{
  if (DeclarationState != External) DeclarationState += IsNested;
  else if (!p) DeclarationState = NoParameters;
  else {
    DeclarationState = HasParameters;
    New region[49]
    Nest[34](‘identifier’,‘1’)
  }
}

void
EndDeclaration()
{
  if (DeclarationState > IsNested) DeclarationState -= IsNested;
  else {
    if (DeclarationState == HasParameters) {
      Restore[36]
      End region[50]
    }
    DeclarationState = External;
  }
}
```

This macro is defined in definitions 51 and 54.
This macro is invoked in definition 55.

1.4 Operational Specifications

Operational specifications of some characteristics are provided directly in a version of C that is compatible with C++.

1.4.1 Phrase.c

This code is provided with controls so that it can be compiled with non-ANSI C compilers.

Phrase.c[55]:

```
#include "eliproto.h"
#include "err.h"
#include "envmod.h"
#include "termcode.h"
#include "pdl_gen.h"
#include "Phrase.h"

BindData    CurrentId;
Environment CurrentEnv;

IdProperties Context = { identifier, 1 };
```

State variable definitions[51]

```
void
#if PROTO_OK
Bind(void)
#else
Bind()
#endif
```
Bind(void)[43]

```
void
#if PROTO_OK
IdnOrType(char *start, int length, int *syncode, int *rep)
#else
IdnOrType(start, length, syncode, rep)
char *start; int length, *syncode; int *rep;
#endif
```
*IdnOrType(char *start, int length, int *syncode, int *rep)*[28]

```
int
#if PROTO_OK
Reparatur(POSITION *coord, int *syncode, int *rep)
#else
Reparatur(coord, syncode, rep) POSITION *coord; int *syncode, *rep;
#endif
```
*Reparatur(POSITION *coord, int *syncode, int *rep)*[29]

```
void
#if PROTO_OK
BeginDeclaration(int c, int b)
#else
BeginDeclaration(c, b) int c, b;
#endif
```
BeginDeclaration(int c, int b)[35]

This macro is attached to a product file.

1.4.2 Phrase.h

Interface specifications for the operational descriptions are controlled so that they may be included several times without the danger of multiple definitions.

Phrase.h[56]:

```
#ifndef PHRASE_H
#define PHRASE_H

#include "eliproto.h"
#include "envmod.h"
#include "RegionStack.h"
```

```
#include "IdStateStack.h"
#include "BindStack.h"
#include "reparatur.h"
```

State variable declarations[30]

```
extern void Bind             ELI_ARG((void));
extern void BeginDeclaration ELI_ARG((int,int));
extern void BeginParameters  ELI_ARG((int));
extern void EndDeclaration   ELI_ARG((void));

#endif
```

This macro is attached to a product file.

1.4.3 Phrase.specs

Phrase.specs[57]:

Instantiate an environment module for identifier analysis[38]
Instantiate a stack module for environment values[48]
Instantiate a stack module for deferred bindings[44]
Instantiate a stack module for state values[33]

This macro is attached to a product file.

1.4.4 Phrase.head

Phrase.head[58]:

```
#include "Phrase.h"
#include "termcode.h"
#include "IdStateStack.h"
```

This macro is attached to a product file.

1.4.5 Phrase.init

Phrase.init[59]:

Reset state[31]
Initialize the set of bindings[40]

This macro is attached to a product file.

Chapter 2

The Abstract Syntax Tree

An abstract syntax tree reflects the semantic structure of a program. Computations over the abstract syntax tree are used to verify context conditions, and to provide information for translating or interpreting the program. The abstract syntax tree should be designed to facilitate these computations.

Eli provides a number of library modules that embody computations useful in the semantic analysis of a variety of languages. These modules are often coupled to an abstract syntax tree by having tree symbols inherit computational roles provided by the module. One aspect of the design of the tree is to ensure that symbols appropriate to these roles are available.

The abstract syntax tree structure is defined by a set of rules written in LIDO. Each rule describes a particular *context* that could appear in an abstract syntax tree representing a C program. It is convenient to have a tree root that is distinguished from the root of the subtree representing the program:

C.lido[1]:

```
RULE: Source ::= program END;
```

AST nodes[2]

This macro is attached to a product file.

2.1 String constants

A string constant may be made up of a concatenation of string literals. Such a concatenation is not relevant for the semantic analysis, but must be carried out if the program is being translated. Thus it is made explicit in the abstract syntax tree.

AST nodes[2]:

```
RULE: StringSeq ::= string_literal           END;
RULE: StringSeq ::= StringSeq string_literal END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.2 Identifiers

An identifier belongs to one of four different name spaces. Section 6.1.2.3 of the standard points out that the syntactic context determines the name space to which a particular identifier occurrence belongs. These contexts are distinguished in the abstract syntax tree by distinct names.

Each of the four name spaces has at least one defining occurrence and one applied occurrence. If an identifier in the label name space happens to also be declared as a `typedef_name` in the name space of ordinary identifiers, the scanner will classify it as a `typedef_name` instead of an `identifier` because the `:` that distinguishes the label context *follows* the identifier. Thus the identifier that is the defining occurrence of a label may have been classified as either a `typedef_name` or an `identifier`. (See Section 1.3 for the details of the classification task.)

Identifiers in the tag name space have more complex semantics, discussed in Section 6.5.2.3 of the standard. Some of the occurrences are always defining, others are always applied, but the characteristics of some may be influenced by constructs remotefrom their position. Further information can be found in Section 2.4.1.

Ordinary identifiers may denote objects or functions on the one hand, and types on the other. The semantics of objects and functions are quite different from the semantics of types, and therefore it is useful to separate object and function identifier occurrences from type identifier occurrences in the tree. Both defining and applied occurrences of type identifiers can be distinguished during scanning, so this is just a matter of assigning them distinct tree nodes.

A `parameter_id` is a defining ocurrence of an ordinary identifier in the parameter list of a function definition. This is a very special context present only to enable compilation of legacy programs.

AST nodes[3]:

```
RULE: LabelDef     ::= identifier   END;
RULE: LabelDef     ::= typedef_name END;
RULE: LabelUse     ::= identifier   END;

RULE: TagDef       ::= identifier   END;
RULE: TagUse       ::= identifier   END;
RULE: ForwardDef   ::= identifier   END;
RULE: ForwardUse   ::= identifier   END;

RULE: MemberIdDef  ::= member_def   END;
RULE: MemberIdUse  ::= identifier   END;

RULE: TypeIdDef    ::= typedef_def  END;
RULE: TypeIdUse    ::= typedef_name END;
RULE: IdDef        ::= identifier   END;
RULE: IdUse        ::= identifier   END;
RULE: parameter_id ::= identifier   END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.3 Expressions

An expression is a combination of operators and operands that yields a value. Operators can be grouped according to the way in which an expression containing them is evaluated, and the constraints that they place upon the operands. Expressions whose operators fall into the same group can be represented by a single context that has the operator as a component. The grouping strategy is a design decision.

Section 6.2.2.1 of the standard defines an *lvalue* as "an expression that designates an object". Except when it is the operand of the `sizeof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is

converted to the value in the designated object. These operators must therefore be distinguished from all others.

The `&` operator does not use the normal operator identification process because it may require construction of a completely new pointer type.

AST nodes[4]:

```
RULE: post_lvalue_opr     ::= '++'     END;
RULE: post_lvalue_opr     ::= '--'     END;

RULE: lvalue_operator     ::= '++'     END;
RULE: lvalue_operator     ::= '--'     END;
RULE: lvalue_operator     ::= 'sizeof' END;

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

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

Separation of this group into three subgroups is based on differing semantics: the value of the expression is taken before the modification with a `post_lvalue_op`, and only the left operand of an `assignment_operator` is treated as an lvalue.

Sections 6.3.13 and 6.3.13 of the standard state that the `&&` and `||` operators guarantee left-to-right evaluation, with a sequence point (a point at which all side effects of previous evaluations are complete – see Section 5.1.2.3 of the standard). For other binary operations, Section 6.3 of the standard says that the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

AST nodes[5]:

```
RULE: logical_operator ::= '&&' END;
RULE: logical_operator ::= '||' END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

All of the remaining operators have the same lvalue, evaluation order, and side-effect properties. Note that no distinction need be made between unary and binary operators here. That distinction is made by the expression context in which the operator appears. For ease of comparison with the standard, rules have been duplicated for operator indications that are both unary and binary (recall that the unary `&` is an `lvalue_operator`, not a `normal_operator`):

AST nodes[6]:

```
RULE: normal_operator ::= '*'  END;
RULE: normal_operator ::= '+'  END;
RULE: normal_operator ::= '-'  END;
RULE: normal_operator ::= '~'  END;
RULE: normal_operator ::= '!'  END;

RULE: normal_operator ::= '*'  END;
RULE: normal_operator ::= '/'  END;
RULE: normal_operator ::= '%'  END;

RULE: normal_operator ::= '+'  END;
RULE: normal_operator ::= '-'  END;

RULE: normal_operator ::= '<<' END;
RULE: normal_operator ::= '>>' END;

RULE: normal_operator ::= '<'  END;
RULE: normal_operator ::= '>'  END;
RULE: normal_operator ::= '<=' END;
RULE: normal_operator ::= '>=' END;

RULE: normal_operator ::= '==' END;
RULE: normal_operator ::= '!=' END;

RULE: normal_operator ::= '&'  END;

RULE: normal_operator ::= '^'  END;

RULE: normal_operator ::= '|'  END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

Given this grouping of operators, the expression contexts are:

AST nodes[7]:

```
RULE: Expression          ::= IdUse                                  END;
RULE: Expression          ::= character_constant                     END;
RULE: Expression          ::= floating_constant                      END;
RULE: Expression          ::= integer_constant                       END;
RULE: Expression          ::= StringSeq                              END;

RULE: Expression          ::= Expression '[' Expression ']'          END;
RULE: Expression          ::= Expression '(' Arguments ')'           END;
RULE: Expression          ::= Expression '.' MemberIdUse             END;
RULE: Expression          ::= DerefExpr  '->' MemberIdUse            END;
RULE: Expression          ::= Expression post_lvalue_opr             END;

RULE: Expression          ::= lvalue_operator Expression             END;
```

```
RULE: Expression          ::= normal_operator Expression               END;
RULE: Expression          ::= 'sizeof' '(' type_name ')'               END;

RULE: Expression          ::= '(' type_name ')' Expression             END;

RULE: Expression          ::= Expression normal_operator Expression    END;

RULE: Expression          ::= Expression logical_operator Expression   END;

RULE: Expression          ::= Expression '?' Expression ':' Expression END;

RULE: Expression          ::= Expression assignment_operator RHSExpr   END;

RULE: Expression          ::= Expression ',' Expression                END;

RULE: constant_expression ::= Expression                               END;
RULE: constant_expression ::=                                          END;

RULE: DerefExpr           ::= Expression                               END;
RULE: RHSExpr             ::= Expression                               END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

An argument to a function call has a proper superset of the semantics of an expression. Therefore it is called out as an additional context:

AST nodes[8]:

```
RULE: Arguments LISTOF Argument   END;
RULE: Argument  ::=    Expression END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.4 Declarations

A declaration begins with a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities being declared. If a type is not completely described by the specifiers, the remaining information is provided by each declarator.

The declaration of a `ForwardTag`, described in Section 6.5.2.3 of the standard, has implications for the name analysis task that are very different from those of any other declaration. Representing this case by its own context simplifies the semantic analysis task.

AST nodes[9]:

```
RULE: declaration     ::=    Specifiers init_declarator_list_opt ';' END;
RULE: declaration     ::=    struct_or_union ForwardDef ';'          END;

RULE: Specifiers      LISTOF Specifier                                END;
```

```
RULE: init_declarator_list_opt
                      LISTOF init_declarator                          END;
RULE: init_declarator ::=    InitDecl                                 END;
RULE: init_declarator ::=    InitDecl '=' initializer                 END;
RULE: InitDecl        ::=    declarator                                END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

Storage specifiers, type specifiers, and type qualifiers are lumped together in this specification because they must be processed in the same way.

AST nodes[10]:

```
RULE: Specifier ::= 'typedef'                 END;
RULE: Specifier ::= 'extern'                  END;
RULE: Specifier ::= 'static'                  END;
RULE: Specifier ::= 'auto'                    END;
RULE: Specifier ::= 'register'                END;
RULE: Specifier ::= 'void'                    END;
RULE: Specifier ::= 'char'                    END;
RULE: Specifier ::= 'short'                   END;
RULE: Specifier ::= 'int'                     END;
RULE: Specifier ::= 'long'                    END;
RULE: Specifier ::= 'float'                   END;
RULE: Specifier ::= 'double'                  END;
RULE: Specifier ::= 'signed'                  END;
RULE: Specifier ::= 'unsigned'                END;
RULE: Specifier ::= struct_or_union_specifier END;
RULE: Specifier ::= enum_specifier            END;
RULE: Specifier ::= TypeIdUse                 END;
RULE: Specifier ::= 'const'                   END;
RULE: Specifier ::= 'volatile'                END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.4.1 Struct and union specifiers

Structure and union specifiers have the same form, being distinguished only by the appearance of the keyword `struct` or the keyword `union`. The presence of a bracketed list declares a new type, which may or may not be given a tag. `TagDef` represents the context in which a tag is given to a new type.

When a bracketed list is not present, the situation is much more complex. Because of the need to declare types that are related cyclically by pointers, tags may be used before they are given specific types. `ForwardDef` (Section 2.4, above) represents a context in which a forward reference represents a type declared in the current scope. `ForwardUse` represents a reference that may be forward (if the identifier appears in a later `TagDef` or `ForwardDef` context in the same scope) or may not (if the identifier does not appear in such a context but has previously appeared in a `TagDef` or `ForwardDef` context visible at this point).

An additional consideration is that the identifier may *never* appear in a `TagDef` or `ForwardDef` context visible atthis point. That situation is not necessarily an error. The type represented by the tag is incomplete (Section 6.1.2.5 of the standard), and as such it may be legally used in any situation that does not require its size.

Clearly the semantics of `TagDef`, `ForwardDef`, and `ForwardUse` are quite different. Representing them by distinct contexts in the tree simplifies the semantic analysis task.

AST nodes[11]:

```
RULE: struct_or_union_specifier
                        ::= struct_or_union TagDef
                                  '{' struct_declaration_list '}'   END;
RULE: struct_or_union_specifier
                        ::= struct_or_union
                                  '{' struct_declaration_list '}'   END;
RULE: struct_or_union_specifier
                        ::= struct_or_union ForwardUse              END;

RULE: struct_or_union    ::=    'struct'                            END;
RULE: struct_or_union    ::=    'union'                             END;

RULE: struct_declaration_list
                        LISTOF struct_declaration                    END;

RULE: struct_declaration ::=    Specifiers struct_declarator_list ';' END;

RULE: struct_declarator_list
                        LISTOF struct_declarator                     END;

RULE: struct_declarator  ::=    member_declarator                   END;
RULE: struct_declarator  ::=    member_declarator ':' constant_expression END;
RULE: struct_declarator  ::=                      ':' constant_expression END;

RULE: member_declarator ::= member_pointer_declarator END;
RULE: member_declarator ::= MemberIdDef END;
RULE: member_declarator ::= member_array_declarator END;
RULE: member_declarator ::= member_function_declarator END;
RULE: member_pointer_declarator ::= '*' Specifiers member_declarator END;
RULE: member_array_declarator ::= member_declarator '[' constant_expression ']' END;
RULE: member_function_declarator ::= member_declarator '(' parameter_type_list ')' END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.4.2 Enumeration specifiers

According to Section 6.5.2.3 of the standard, there is no forward use of an enumerator tag similar to the ones discussed in Section 2.4.1. The reason is that enumerations do not involve any requirement for cyclic definitions. It is always possible to simply declare an enumeration before it is used. This means that enumerations involve only the `TagDef` and`TagUse` contexts.

AST nodes[12]:

```
RULE: enum_specifier  ::=    'enum' TagDef '{' enumerator_list '}'        END;
RULE: enum_specifier  ::=    'enum'        '{' enumerator_list '}'        END;
```

```
RULE: enum_specifier  ::=    'enum' TagUse                                    END;

RULE: enumerator_list LISTOF enumerator                                       END;
RULE: enumerator      ::=    enumeration_constant                             END;
RULE: enumerator      ::=    enumeration_constant '=' constant_expression END;
RULE: enumeration_constant
                      ::=    identifier                                       END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.4.3 Declarators

Each declarator declares one identifier in the member or ordinary identifier name space (Section 6.1.2.3 of the standard). An ordinary identifier may be either a type identifier or an identifier denoting an object or function. All three of these possibilities are given distinct contexts because they have distinct semantics.

The declarator may also create a type for the identifier it is declaring by modifying the type described by the `Specifier` in the containing declaration. Type creation is represented in the tree by the `pointer_declarator`, `array_declarator`, and `function_declarator` contexts.

Two contexts are distinguished for prarmeters: those definitely belonging to a function prototype, and those possibly belonging to a function definition. Parameters possible belonging to a function definition have different scopes than those belonging to a function prototype, and also parameters belonging to a function definition may be simple identifiers rather than declarations.

AST nodes[13]:

```
RULE: declarator      ::= TypeIdDef                                           END;
RULE: declarator      ::= IdDef                                               END;
RULE: declarator      ::= pointer_declarator                                  END;
RULE: declarator      ::= array_declarator                                    END;
RULE: declarator      ::= function_declarator                                 END;

RULE: pointer_declarator  ::= '*' Specifiers declarator                       END;
RULE: array_declarator    ::= declarator '[' constant_expression ']'        END;
RULE: function_declarator ::= declarator '(' parameter_type_list ')'       END;
RULE: function_declarator ::= declarator '(' parameters ')'                 END;

RULE: parameter_type_list       LISTOF ParameterType  | DotDotDot END;
RULE: parameters LISTOF parameter_id | ParameterType  | DotDotDot END;

RULE: ParameterType         ::= parameter_declaration                       END;

RULE: parameter_declaration ::= Specifiers ParameterDecl                    END;
RULE: parameter_declaration ::= Specifiers abstract_declarator             END;
RULE: parameter_declaration ::= Specifiers                                  END;
RULE: DotDotDot             ::= '...'                                       END;
RULE: ParameterDecl         ::= declarator                                  END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.4.4 Type names

A type name is used to denote a type. If it is not possible to describe the desired type with specifiers alone, then a declarator is needed. However, that declarator is not allowed to declare an identifier. This situation, a declarator that describes a type but does not declare an identifier, is represented by the `abstract_declarator` contexts.

AST nodes[14]:

```
RULE: type_name                   ::= Specifiers                          END;
RULE: type_name                   ::= Specifiers abstract_declarator      END;

RULE: abstract_declarator         ::= pointer_abstract_declarator         END;
RULE: abstract_declarator         ::= array_abstract_declarator           END;
RULE: abstract_declarator         ::= function_abstract_declarator        END;

RULE: pointer_abstract_declarator ::= '*' Specifiers                      END;
RULE: pointer_abstract_declarator ::= '*' Specifiers abstract_declarator  END;

RULE: array_abstract_declarator   ::= '[' constant_expression ']'         END;
RULE: array_abstract_declarator   ::=
                  abstract_declarator '[' constant_expression ']'         END;

RULE: function_abstract_declarator ::= '(' parameter_type_list ')'        END;
RULE: function_abstract_declarator ::=
                  abstract_declarator '(' parameter_type_list ')'         END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.4.5 Initialization

AST nodes[15]:

```
RULE: initializer       ::=    Expression                      END;
RULE: initializer       ::=    Row_initializer                 END;
RULE: Row_initializer   LISTOF initializer | Empty_initializer END;
RULE: Empty_initializer ::=                                    END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.5 Statements

AST nodes[16]:

```
RULE: statement ::= LabelDef ':' statement                              END;
RULE: statement ::= 'case' constant_expression ':' statement            END;
RULE: statement ::= 'default' ':' statement                             END;
```

```
RULE: statement ::= compound_statement                                    END;
RULE: compound_statement
                ::= '{' body '}'                                          END;
RULE: body   LISTOF declaration | statement                               END;

RULE: statement ::= Expression ';'                                        END;
RULE: statement ::=            ';'                                        END;

RULE: statement ::= 'if' '(' Expression ')' statement 'else' statement    END;
RULE: statement ::= 'if' '(' Expression ')' statement                     END;
RULE: statement ::= switch_statement                                      END;
RULE: switch_statement ::= 'switch' '(' Expression ')' statement          END;

RULE: statement ::= iteration_statement                                   END;
RULE: iteration_statement
                ::= 'while' '(' Expression ')' statement                  END;
RULE: iteration_statement
                ::= 'do' statement 'while' '(' Expression ')' ';'         END;
RULE: iteration_statement
                ::= 'for' '(' Exp1or3 ';' Exp2 ';' Exp1or3 ')' statement  END;

RULE: Exp1or3   ::= Expression                                            END;
RULE: Exp1or3   ::=                                                       END;
RULE: Exp2      ::= Expression                                            END;
RULE: Exp2      ::=                                                       END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

Section 6.6.5.3 of the standard says that both the initializer and the step Of a `for` statement may be omitted. Each is then evaluated as a void expression. An omitted condition is replaced by a nonzero constant. These distinct semantics are embodied in specific tree contexts.

AST nodes[17]:

```
RULE: statement ::= 'goto' LabelUse     ';' END;
RULE: statement ::= 'continue'          ';' END;
RULE: statement ::= 'break'             ';' END;
RULE: statement ::= 'return'            ';' END;
RULE: statement ::= 'return' Expression ';' END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.6 External definitions

AST nodes[18]:

```
RULE: program LISTOF declaration | function_definition         END;

RULE: function_definition ::=
```

```
        Specifiers declaration_list FunctionDecl function_body END;
RULE: function_body       ::=    '{' body '}'                  END;
RULE: FunctionDecl        ::=    declarator                    END;

RULE: declaration_list    LISTOF par_declaration               END;
RULE: par_declaration     ::=    Specifiers par_id_decls ';'   END;
RULE: par_id_decls        LISTOF ParameterTypeId               END;
RULE: ParameterTypeId     ::=    ParameterDecl                 END;
```

This macro is defined in definitions 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 1.

2.7 Tree.map

An abstract syntax tree does not distinguish symbols of the original grammar that are semantically identical. If a symbol in the abstract syntax tree corresponds to a number of distinct grammar symbols, that correspondence must be given as part of the specification. This is the role of a file of type `map`:

Tree.map[19]:

```
MAPSYM

parameter_type_list ::=
  empty_parameter_type_list .

Arguments ::= argument_expression_list_opt .

statement ::= selection_statement expression_statement jump_statement
  labeled_statement .

Expression ::=
  constant
  primary_expression postfix_expression unary_expression cast_expression
  multiplicative_expression additive_expression shift_expression
  relational_expression equality_expression AND_expression
  exclusive_OR_expression inclusive_OR_expression logical_AND_expression
  logical_OR_expression conditional_expression assignment_expression
  expression .

constant_expression ::= constant_exp_opt .

Exp1or3 ::=
  expression_opt1 .
Exp2 ::=
  expression_opt2 .

normal_operator ::= unary_operator .

Specifiers ::=
  declaration_specifiers empty_declaration_specifiers
  type_qualifier_list_opt specifier_qualifier_list .
```

```
Specifier ::=
  storage_class_specifier
  type_qualifier .

declarator ::=
  direct_declarator .

member_declarator ::=
  member_direct_declarator .

abstract_declarator ::= direct_abstract_declarator .

MAPRULE

function_definition:
  empty_declaration_specifiers declarator declaration_list function_body
  < $1 $3 $2 $4 > .

function_definition:
  declaration_specifiers declarator declaration_list function_body
  < $1 $3 $2 $4 > .
```

This macro is attached to a product file.

Chapter 3

Name Analysis

Name analysis establishes a meaning (possibly "no meaning", if certain errors have been made) for each identifier in the program. In the process, it decorates the nodes of the abstract syntax tree with attributes and stores properties of entities in the definition table. These attributes and properties provide the basis for constraint checking, and can also be used to guide a translation of the program.

Every identifier occurrence requires an integer-valued `Sym` attribute specifying its string table index. The string table index is provided by the lexical analyzer as the value of the terminal symbol `identifer` (see Section 1.1.3). `IdentOcc` defines a computation setting the `Sym` attribute from the terminal symbol value; that computation can be inherited by any nonterminal representing an identifier occurrence:

Name.lido[1]:

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

Regions of text encompassing identifier scopes[4]
Identifier occurrences[5]
Checking for definition errors[8]

This macro is attached to a product file.

Many of the computations used during semantic analysis are common to many trees, and have been captured in library modules. Modules actually used in the semantic analysis of C programs must be instantiated:

Name.specs[2]:

Instantiate modules[3]

This macro is attached to a product file.

3.1 Name spaces and scopes

Section 6.1.2.3 of the standard defines four *name spaces* for identifiers in C, and therefore we use four instances of the necessary environment modules:

Instantiate modules[3]:

```
$/Name/AlgScope.gnrc      +instance=Label  :inst
$/Name/AlgScope.gnrc      +instance=Tag    :inst
$/Name/AlgRangeSeq.gnrc   +instance=Tag    :inst
$/Name/AlgScope.gnrc      +instance=Member :inst
$/Name/ScopeProp.gnrc     +instance=Member :inst
$/Name/CScope.gnrc                         :inst
$/Name/CRangeSeq.gnrc                      :inst
```

This macro is defined in definitions 3 and 9.
This macro is invoked in definition 2.

(The unnamed instances implement the name space of "ordinary identifiers". An ordinary identifier is any identifier that does not belong to one of the other name spaces.)

Label identifiers obey ALGOL-like scope rules because Section 6.1.2.1 of the standard states explicitly that the scope of a label identifier is the entire function in which it appears.

Section 6.5.2.3 of the standard describes the behavior of tags, and although that description is couched in terms of the sequence in which constructs appear in the program, the overall effect is that tags obey ALGOL-like scope rules.

Member identifier declarations do not interact with any other identifier occurrences in their scopes. Similarly, applied occurrences of member identifiers don't interact with any other identifier occurrences. Thus member identifiers could be considered to obey either ALGOL-like or C-like scope rules. We chose to use ALGOL-like scope rules for compatibility with the `ScopeProp` module.

Ordinary identifiers obey C-like scope rules, in which the scope begins with the declaration of the identifier and ends at the end of some source language construct.

Section 6.1.2.1 of the standard defines the points at which the scopes of various identifiers *terminate*, and then states that two identifiers have the same scope if and only if their scopes terminate at the same point. Certain symbols of Section 2's abstract syntax tree correspond to constructs ending at those termination points and encompassing all of the possible points at which the scope of an identifier could begin. Each of these symbols inherits the `RangeScope` role of the corresponding name space module.

Identifiers appearing within the list of parameter declarations in a function definition have block scope terminating at the } that closes the function body. Although the parameter list and the function body are represented by nodes that have a common parent (`function_definition`), there are intervening nodes. This requires use of the range sequence module, with `function_definition` inheriting the `RangeSequence` role and the parameter list and function body each inheriting the `RangeElement` role. The definition of the range sequence module requires that `RangeElement` be inherited by a node belonging to a subtree rooted in a node that inherits `RangeSequence`.

The `parameters` node, which represents the parameter list of a function definition, is a component of a `declarator`, which occurs in contexts other than function definitions. When `declarator` occurs a context other than a function definition, `parameters` represents a function prototype scope. There is no practical way to make a syntactic distinction between the use of a `declarator` in a function definition and the use of a `declarator` in other contexts.

Since `parameters` must inherit `RangeElement` in order to implement the block scope of a function definition, it must *always* appear in a subtree rooted in a node inheriting `RangeSequence`. Thus several other nodes that would not normally be considered to represent constructs significant for name analysis inherit that role.

Regions of text encompassing identifier scopes[4]:

```
SYMBOL function_body          INHERITS LabelRangeScope                END;
```

```
SYMBOL function_definition     INHERITS TagRangeSequence                END;
SYMBOL parameters               INHERITS TagRangeElement                 END;
SYMBOL function_body           INHERITS TagRangeElement                 END;
SYMBOL compound_statement      INHERITS TagRangeScope                   END;
SYMBOL Prototype                INHERITS TagRangeScope                   END;

SYMBOL struct_declaration_list INHERITS MemberExportRange               END;

SYMBOL function_definition     INHERITS RangeSequence                   END;
SYMBOL parameters               INHERITS RangeElement                    END;
SYMBOL declaration_list        INHERITS RangeElement                    END;
SYMBOL function_body           INHERITS RangeElement                    END;
SYMBOL compound_statement      INHERITS RangeScope                      END;
SYMBOL parameter_type_list     INHERITS RangeScope                      END;

SYMBOL init_declarator         INHERITS RangeSequence, TagRangeSequence END;
SYMBOL parameter_declaration   INHERITS RangeSequence, TagRangeSequence END;
```

This macro is invoked in definition 1.

3.2 Identifier occurrences

Section 6.1.2.1 of the standard says that tags are bound just after their appearance in the text, enumeration constants are bound after their enumerator, and any other identifier is bound after its declarator. This means that the `IdDefScope` computational role is sometimes inherited by an identifier occurrence, and sometimes by a larger construct with an identifier occurrence as a component.

The client of the name analysis modules must compute a `Sym` attribute for every node whose left-hand-side symbol inherits `IdDefScope`; `IdDefScope` attaches a `Bind` attribute and a `Key` attribute to that node.

Identifier occurrences that are components of larger constructs binding those identifiers inherit the `IdInDeclarator` role, which both acts as a source for the larger construct's `Sym` attribute and establishes a `Key` attribute for the identifier occurrence. The larger construct must inherit the `DeclaratorWithId` role, and there must be a 1-1 correspondence between a symbol inheriting `DeclaratorWithId` and one of its children that inherits `IdInDeclarator`.

Identifier occurrences[5]:

```
SYMBOL LabelDef      INHERITS IdentOcc, LabelIdDefScope  END;
SYMBOL LabelUse      INHERITS IdentOcc, LabelIdUseEnv    END;

SYMBOL TagDef        INHERITS IdentOcc, TagIdDefScope    END;
SYMBOL TagUse        INHERITS IdentOcc, TagIdUseEnv      END;
SYMBOL ForwardDef    INHERITS IdentOcc, TagIdDefScope    END;
SYMBOL ForwardUse    INHERITS IdentOcc, TagIdUseEnv      END;

SYMBOL MemberIdDef   INHERITS IdentOcc, MemberIdDefScope END;
SYMBOL MemberIdUse   INHERITS IdentOcc, MemberQualIdUse  END;

SYMBOL TypeIdDef     INHERITS IdentOcc, IdInDeclarator   END;
SYMBOL TypeIdUse     INHERITS IdentOcc, IdUseEnv         END;

SYMBOL InitDecl      INHERITS           DeclaratorWithId END;
```

```
SYMBOL FunctionDecl  INHERITS           DeclaratorWithId END;
SYMBOL ParameterDecl INHERITS           DeclaratorWithId END;
SYMBOL enumerator    INHERITS           DeclaratorWithId END;
SYMBOL enumeration_constant
                     INHERITS IdentOcc, IdInDeclarator   END;
SYMBOL IdDef         INHERITS IdentOcc, IdInDeclarator   END;
SYMBOL IdUse         INHERITS IdentOcc, IdUseEnv         END;
SYMBOL parameter_id  INHERITS IdentOcc, IdUseEnv         END;
```

This macro is defined in definitions 5, 6, and 7.
This macro is invoked in definition 1.

A `MemberIdUse` is bound in the scope of the structure or union resulting from the expression preceeding the access operator:

Identifier occurrences[6]:

```
SYMBOL struct_declaration_list COMPUTE
  INH.MemberScopeKey=THIS.Type;
END;

SYMBOL MemberDecl INHERITS DeclaratorWithId COMPUTE
  SYNT.Sym=NoStrIndex;
END;
```

This macro is defined in definitions 5, 6, and 7.
This macro is invoked in definition 1.

The behavior of `ForwardUse` described in Section 6.5.2.3 does not quite fit the `IdUseEnv` computational roles of the `AlgScope` module: There may be *no* defining occurrences of tag in its environment, but nevertheless it should have a binding. This can be arranged by creating a default binding in the root scope:

Identifier occurrences[7]:

```
RULE: ForwardUse ::= identifier
COMPUTE
  ForwardUse.GotDefaultBinding=
    BindIdn(INCLUDING Source.TagEnv,identifier);
END;

SYMBOL Source COMPUTE
  SYNT.TagGotKeys=
    CONSTITUENTS ForwardUse.GotDefaultBinding <- THIS.TagGotLocKeys;
END;
```

This macro is defined in definitions 5, 6, and 7.
This macro is invoked in definition 1.

3.3 Checking for definition errors

According to Section 6.1.2.1 of the standard, an identifier can be used only within its scope. If an applied occurrence of an identifier lies outside of any scope of a defining occurrence of that identifier, the name

analysis module will not be able to bind it. The module provide roles that can be inherited by symbols representing applied occurrences of identifiers in order to report such errors:

Checking for definition errors[8]:

```
SYMBOL LabelUse    INHERITS       ChkIdUse     END;
SYMBOL TagUse      INHERITS       ChkIdUse     END;
SYMBOL IdUse       INHERITS       ChkIdUse     END;
SYMBOL MemberIdUse INHERITS MemberChkQualIdUse END;
```

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

Section 6.1.2.1 of the standard says that label names must be unique within a function. Section 6.5 of the standard says that if an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space (the standard makes an exception for tags in Section 6.5.2.3 to allow for forward declarations).

Instantiate modules[9]:

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

This macro is defined in definitions 3 and 9.
This macro is invoked in definition 2.

Checking for definition errors[10]:

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

SYMBOL LabelDef             INHERITS MultDefChk         END;
SYMBOL TagDef               INHERITS MultDefChk         END;
SYMBOL MemberIdDef          INHERITS MultDefChk         END;
SYMBOL enumeration_constant INHERITS MultDefChk         END;
SYMBOL TypeIdDef            INHERITS MultDefChk         END;
```

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

Chapter 4

Type Analysis

Type analysis assigns a type to every value in the program.

Type.lido[1]:

```
ATTR Type: DefTableKey; /* C type associated with every value */
```

Create standard pointer types[6]
Associate types with identifiers[30]
Declaration specifiers[34]
Relationships among types[47]
Operator identification[53]

This macro is attached to a product file.

Type.pdl[2]:

Define keys, properties, and access functions[48]

This macro is attached to a product file.

Type.specs[3]:

Instantiate modules[29]

This macro is attached to a product file.

4.1 The C type system

The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression: the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).

This section uses Eli's *O*perator *I*dentification *L*anguage OIL to formalize the standard's description of types (Section 4.1.1), conversions among types (Section 4.1.2) and type constraints on operators (Section 4.1.3).

Type.oil[4]:

Types[5]
Integral promotions[13]
Usual arithmetic conversions[14]
Define a scalar type[16]
void[17]
The null pointer constant[18]
Operators[19]

This macro is attached to a product file.

4.1.1 Types

Section 6.1.2.5 of the standard describes all of the C types. This section introduces the OIL identifiers corresponding to the standard's names for basic C types, sets of types, and user-defined types. All of these identifiers begin with `TypeIs_`. Identifiers representing the basic types of C continue with one or more C keywords, all lower case, giving the canonic name of the type in the standard (e.g. `TypeIs_char`). Identifiers representing sets of types and user-defined types continue with a capitalized name. That name is the one used by the standard to describe the set of types or the derivation of the user-defined type wherever possible (e.g. `TypeIs_Integral`, `TypeIs_Pointer`).

Sets of types are specified in this section by OIL `SET` directives.

Types[5]:

```
SET TypeIs_Signed_Integer=
  [TypeIs_signed_char, TypeIs_short, TypeIs_int, TypeIs_long];

SET TypeIs_Unsigned_Integer=
  [TypeIs_unsigned_char, TypeIs_unsigned_short, TypeIs_unsigned_int,
   TypeIs_unsigned_long];

SET TypeIs_Floating=
  [TypeIs_float, TypeIs_double, TypeIs_long_double];
```

This macro is defined in definitions 5, 10, 11, 12, and 15.
This macro is invoked in definition 4.

Each standard type has an associated pointer type, which may or may not be declared by the programmer. The pointer types must be defined, however, in order to be available as the result of an `&` operator applied to a standard type:

Create standard pointer types[6]:

```
SYMBOL Source COMPUTE
  SYNT.GotType=BasicPointerTypes();
  SYNT.GotOper=BasicPointerRefs() <- THIS.GotAllTypes;
END;
```

This macro is invoked in definition 1.

void BasicPointerTypes(void)[7]:

```
{ int i;

  for (i = 0; i < COUNT; i++) {
    Pointer[i] = NewType();
    ResetTypeName(Pointer[i], Name[i]);
    AddTypeToBlock(
      Pointer[i],
      PointerTypes,
      SingleDefTableKeyList(Type[i]));
  }
}
```

This macro is invoked in definition 55.

void BasicPointerRefs(void)[8]:

```
{ int i;

  for (i = 0; Type[i] != NoKey; i++) {
    InstClass1(TypeIs_Pointer,FinalType(Pointer[i]),Type[i]);
  }
}
```

This macro is invoked in definition 55.

Two arrays of the same length are common to the two routines:

Type key arrays[9]:

```
#define COUNT 12

static DefTableKey Pointer[COUNT], Type[] = {
  TypeIs_signed_char,
  TypeIs_short,
  TypeIs_int,
  TypeIs_long,
  TypeIs_unsigned_char,
  TypeIs_unsigned_short,
  TypeIs_unsigned_int,
  TypeIs_unsigned_long ,
  TypeIs_float,
  TypeIs_double,
  TypeIs_long_double };

static char *Name[] = {
  "signed_char*",
  "short*",
  "int*",
  "long*",
  "unsigned_char*",
  "unsigned_short*",
  "unsigned_int*",
  "unsigned_long*",
```

```
  "float*",
  "double*",
  "long_double*" };
```

This macro is invoked in definition 55.

Section 6.1.2.5 of the standard provides constructors enabling a user to define additional types, called *derived types*. It gives a short description of each constructor, explaining the relationship (if any) between the type derived and other types.

Each of these constructors is specified in this section by an OIL `CLASS`. An Oil `CLASS` is a parameterized template for construction of a new type and all of the related operators. The standard gives no information about the operators related to a type in Section 6.1.2.5. This specification follows the standard, using brief descriptions of the operators here and actually defining them in other sections. For each kind of operator, the description can be found via the cross reference on the brief description.

Types[10]:

```
CLASS TypeIs_Enum() BEGIN
COERCION
  (TypeIs_Enum): TypeIs_int;
OPER
  Enum_Assign_Op(TypeIs_Enum, TypeIs_int): TypeIs_Enum;
END;

CLASS TypeIs_Array(elementType, pointerType) BEGIN
COERCION
  (TypeIs_Array): pointerType;
  (TypeIs_Array): TypeIs_void;
OPER
  Array_Subscript_Op(TypeIs_Array, TypeIs_unsigned_long): elementType;
END;

CLASS TypeIs_Struct() BEGIN
COERCION
  (TypeIs_Struct): TypeIs_void;
OPER
  Struct_Assign_Op(TypeIs_Struct, TypeIs_Struct): TypeIs_Struct;
END;

CLASS TypeIs_Union() BEGIN
COERCION
  (TypeIs_Union): TypeIs_void;
OPER
  Union_Assign_Op(TypeIs_Union, TypeIs_Union): TypeIs_Union;
END;

CLASS TypeIs_Function(fnptrType) BEGIN
COERCION
  (TypeIs_Function): fnptrType;
  (TypeIs_Function): TypeIs_void;
END;
```

```
CLASS TypeIs_Pointer(referencedType) BEGIN
COERCION
  (TypeIs_Pointer): TypeIs_void;
  (TypeIs_Pointer): TypeIs_VoidPointer;
  (TypeIs_NULL):    TypeIs_Pointer;
OPER
  Subscript_Op(TypeIs_Pointer, TypeIs_unsigned_long): referencedType;
  Ptr_Inc_Op, Ptr_Dec_Op(TypeIs_Pointer): TypeIs_Pointer;
  Ptr_Deref_Op(TypeIs_Pointer): referencedType;
  Ptr_Ref_Op  (referencedType): TypeIs_Pointer;
  Cast_IntegraltoPtr(TypeIs_unsigned_long): TypeIs_Pointer;
  Cast_VoidPtrtoPtr (TypeIs_VoidPointer):   TypeIs_Pointer;
  Ptr_Add_Op, Ptr_Sub_Op(TypeIs_Pointer, TypeIs_unsigned_long): TypeIs_Pointer;
  Ptr_Rev_Add_Op        (TypeIs_unsigned_long, TypeIs_Pointer): TypeIs_Pointer;
  Ptr_Ptr_Sub_Op        (TypeIs_Pointer, TypeIs_Pointer): TypeIs_unsigned_long;
  Ptr_LT_Op, Ptr_GT_Op, Ptr_LTE_Op,
  Ptr_GTE_Op(TypeIs_Pointer, TypeIs_Pointer): TypeIs_int;
  Ptr_Eq_Op, Ptr_NEq_Op(TypeIs_Pointer, TypeIs_Pointer): TypeIs_int;
  Ptr_Cond_Op(TypeIs_Scalar,TypeIs_Pointer,TypeIs_Pointer): TypeIs_Pointer;
  Ptr_Assign_Op     (TypeIs_Pointer, TypeIs_Pointer):     TypeIs_Pointer;
  Ptr_Void_Assign_Op(TypeIs_Pointer, TypeIs_VoidPointer): TypeIs_Pointer;
  Ptr_Plus_Eq_Op (TypeIs_Pointer, TypeIs_unsigned_long): TypeIs_Pointer;
  Ptr_Minus_Eq_Op(TypeIs_Pointer, TypeIs_unsigned_long): TypeIs_Pointer;
END;
```

This macro is defined in definitions 5, 10, 11, 12, and 15.
This macro is invoked in definition 4.

Section 6.1.2.5 of the standard says that `char`, the signed and unsigned integer types, and the enumerated types are called *integral types*. Integral and floating types are collectively called *arithmetic types*. Arithmetic and pointer types are collectively called *scalar types*. Unfortunately, an OIL set is a static object and therefore cannot contain a class as a member.

Types[11]:

```
SET TypeIs_Integral =
  [TypeIs_char] + TypeIs_Signed_Integer + TypeIs_Unsigned_Integer;

SET TypeIs_Arithmetic = TypeIs_Integral + TypeIs_Floating;

SET TypeIs_Scalar = TypeIs_Arithmetic + [TypeIs_VoidPointer];
```

This macro is defined in definitions 5, 10, 11, 12, and 15.
This macro is invoked in definition 4.

Section 6.1.4 of the standard describes string literals. The semantics of a string literal are exactly those of a pointer to a character sequence that is terminated by a zero byte:

Types[12]:

```
COERCION
  (TypeIs_string): TypeIs_void;
  (TypeIs_string): TypeIs_VoidPointer;
OPER
  IndexString(TypeIs_string, TypeIs_unsigned_long): TypeIs_char;
  DerefString(TypeIs_string): TypeIs_char;
```

This macro is defined in definitions 5, 10, 11, 12, and 15.
This macro is invoked in definition 4.

4.1.2 Conversions

Section 6.2 of the standard describes the relationships between types that are engendered by *implicit conversions*. Such implicit conversions are defined in OIL by *coercions*.

Implicit conversions between derived types can only be defined after the types themselves are defined. Thus coercions involving derived types must be components of the relevant OIL classes. These coercions are defined here as macros that are invoked as components of the classes defined in Section 4.1.1.

Integral promotions[13]:

```
COERCION
  CChartoInt          (TypeIs_char):           TypeIs_int;
  CUnsignedChartoInt  (TypeIs_unsigned_char):  TypeIs_int;
  CShorttoInt         (TypeIs_short):          TypeIs_int;
  CUnsignedShorttoInt (TypeIs_unsigned_short): TypeIs_int;
```

This macro is invoked in definition 4.

Usual arithmetic conversions[14]:

```
COERCION
  CDoubletoLongDouble (TypeIs_double):         TypeIs_long_double;
  CFloattoDouble      (TypeIs_float):          TypeIs_double;
  CUnsignedLongtoFloat(TypeIs_unsigned_long):  TypeIs_float;

  CLongtoUnsignedLong (TypeIs_long):           TypeIs_unsigned_long;
  CUnsignedInttoLong  (TypeIs_unsigned_int):   TypeIs_long;
  CInttoLong          (TypeIs_int):            TypeIs_long;
  CInttoUnsignedInt   (TypeIs_int):            TypeIs_unsigned_int;
```

This macro is invoked in definition 4.

Certain operators require that the integral promotion be performed on their operand(s), and the result has the promoted type. These operators are therefore defined in terms of sets that include only promoted integer types:

Types[15]:

```
SET TypeIs_IntegralPromoted =
  [TypeIs_int, TypeIs_unsigned_int, TypeIs_long, TypeIs_unsigned_long];

SET TypeIs_ArithPromoted    =
  TypeIs_IntegralPromoted + TypeIs_Floating;
```

This macro is defined in definitions 5, 10, 11, 12, and 15.
This macro is invoked in definition 4.

A conditional can accept any scalar type. Thus we need a type key representing any scalar type:

Define a scalar type[16]:

```
COERCION (TypeIs_Scalar):scalarType;
```

This macro is invoked in definition 4.

Section 6.2.2.1 of the standard says that an lvalue that has type "array of *type*" is converted to an expression that has type "pointer to *type*" except when it is the operand of `sizeof` or the unary `&` operator, or is a character string literal used to initialize an array. A function designator of type "function returning *type*" is converted to an expression that has type 'pointer to function returning *type*" under similar circumstances:

Section 6.2.2.2 says that an expression occurs in a context where a void expression is required, its value or designator is discarded.

void[17]:

```
COERCION CScalartoVoid   (TypeIs_Scalar):   TypeIs_void;
```

This macro is invoked in definition 4.

Section 6.2.2.3 of the standard describes the conversions among pointers. `TypeIs_NULL` represents an integral constant expression with the value 0, which can be converted to a pointer of any type.

`CNulltoVoidPtr` is required because it is possible to use an integral constant expression with the value 0 as a value of type `void*` directly, and `CNulltoIntegral` is required to allow an integral constant expression with the value 0 to be interpreted as an integral value rather than a null pointer.

The null pointer constant[18]:

```
COERCION
  (TypeIs_NULL): TypeIs_Integral;
OPER
  CNulltoVoidPtr (TypeIs_NULL): TypeIs_VoidPointer;
INDICATION
  Cast_Indication: CNulltoVoidPtr;
```

This macro is invoked in definition 4.

4.1.3 Constraints on operators

Many of the subsections of Section 6.3 of the standard describe constraints on the types of operands that a given operator can have. This section formalizes those constraints, defining each C operator by an `INDICATION`. The constraints are described by a set of `OPER`ators, each with a specific type signature. Each `INDICATION` is associated with a comma-separated list of the `OPER`ators that satisfy the standard's constraints on that C operator.

Operators[19]:

```
INDICATION
  Subscript_Indication: Subscript_Op, Array_Subscript_Op, IndexString;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

Distinct subscripting operations are defined for each array type and each pointer type other than `TypeIs_VoidPointer`. The second operand is specified to be a `TypeIs_unsigned_long`, although the standard specifies that it has integral type. Any integral type can be converted to `TypeIs_unsigned_long` by means of implicit conversions, so this specification is equivalent to that of the standard. The main reason for using `TypeIs_unsigned_long` is that OIL does not permit sets as operand specifications within a class definition, but a secondary reason is that this approach reduces the total number of operators in the compiler's database.

Operators[20]:

```
INDICATION
  Plus_Plus_Indication:   Increment_Op, Ptr_Inc_Op;
  Minus_Minus_Indication: Decrement_Op, Ptr_Dec_Op;

OPER
  Increment_Op, Decrement_Op(TypeIs_Arithmetic): TypeIs_Arithmetic;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

Operators[21]:

```
INDICATION
  Star_Indication:  Ptr_Deref_Op, DerefString;
  Amper_Indication: Ptr_Ref_Op;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

Operators[22]:

```
INDICATION
  Plus_Indication:   Plus_Op;
  Minus_Indication:  Minus_Op;
  Tilde_Indication:  Bitwise_Not_Op;
  Bang_Indication:   Logical_Not_Op;
  Sizeof_Indication: Sizeof_op;
  Cast_Indication:   Cast_Op, Cast_IntegraltoPtr, Cast_VoidPtrtoPtr,
                     Cast_VoidVoid, CScalartoVoid;

SET TypeIs_CastResult = TypeIs_Scalar;

OPER
  Plus_Op, Minus_Op(TypeIs_ArithPromoted):    TypeIs_ArithPromoted;
```

```
  Bitwise_Not_Op   (TypeIs_IntegralPromoted): TypeIs_IntegralPromoted;
  Logical_Not_Op   (TypeIs_Scalar):           TypeIs_int;
/* FIXME: sizeof is more constrained than this.  Also, it has different
  semantics in that the operand is not evaluated.  May need another kind
  of operator.
*/
  Sizeof_op        (TypeIs_void):             TypeIs_int;
  Cast_Op          (TypeIs_Scalar):           TypeIs_CastResult;
  Cast_VoidVoid    (TypeIs_void):             TypeIs_void;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

By defining `TypeIs_CastResult` as being equal to `TypeIs_Scalar` and then using these two set identifiers in defining `Cast_Op`, the specification defines a cast from every element of `TypeIs_Scalar` to every other element of `TypeIs_Scalar`. If `Cast_Op` were defined with the signature `(TypeIs_Scalar): TypeIs_Scalar` then the operand and result would be constrained to be identical.

The definition of `Cast_Op` allows any pointer type to be cast to any integral type: There is an implicit conversion from any pointer type to `TypeIs_VoidPointer`, which is an element of `TypeIs_Scalar`, and every integral type is an element of `TypeIs_CastResult`.

Distinct cast operators are defined for each pointer type other than `TypeIs_VoidPointer` to handle casts from arbitrary integers and from `TypeIs_VoidPointer` to that pointer type. The operand of the former is specified to be a `TypeIs_unsigned_long`, although the standard specifies that it has integral type. Any integral type can be converted to `TypeIs_unsigned_long` by means of implicit conversions, so this specification is equivalent to that of the standard. The main reason for using `TypeIs_unsigned_long` is that OIL does not permit sets as operand specifications within a class definition, but a secondary reason is that this approach reduces the total number of operators in the compiler's database.

The implicit conversion from any pointer type to `TypeIs_VoidPointer`, combined with `Cast_VoidPtrtoPtr`, allows any pointer type to be cast to any other pointer type. (This includes pointers to function types.)

Operators[23]:

```
INDICATION
  Star_Indication:    MulOp;
  Slash_Indication:   DivOp;
  Percent_Indication: ModOp;
  Plus_Indication:    AddOp,
                      Void_Ptr_Add_Op, Void_Ptr_Rev_Add_Op,
                      Ptr_Add_Op,      Ptr_Rev_Add_Op;
  Minus_Indication:   SubOp,
                      Void_Ptr_Sub_Op, Ptr_Sub_Op, Ptr_Ptr_Sub_Op;

OPER
  MulOp, DivOp,
  AddOp, SubOp   (TypeIs_Arithmetic, TypeIs_Arithmetic): TypeIs_Arithmetic;
  ModOp          (TypeIs_Integral, TypeIs_Integral):     TypeIs_Integral;
  Void_Ptr_Add_Op,
  Void_Ptr_Sub_Op(TypeIs_VoidPointer, TypeIs_unsigned_long): TypeIs_VoidPointer;
  Void_Ptr_Rev_Add_Op
                 (TypeIs_unsigned_long, TypeIs_VoidPointer): TypeIs_VoidPointer;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

Operators[24]:

```
INDICATION
  Less_Less_Indication:       Bit_Shift_Left_Op;
  Greater_Greater_Indication: Bit_Shift_Right_Op;
  Less_Indication:            LessThan_Op,       Ptr_LT_Op;
  Greater_Indication:         Greater_Op,        Ptr_GT_Op;
  Less_Equal_Indication:      LessThan_Equal_Op, Ptr_LTE_Op;
  Greater_Equal_Indication:   Greater_Equal_Op,  Ptr_GTE_Op;

SET TypeIs_ShiftCount = TypeIs_IntegralPromoted;

OPER
  Bit_Shift_Right_Op,
  Bit_Shift_Left_Op
    (TypeIs_IntegralPromoted, TypeIs_ShiftCount): TypeIs_IntegralPromoted;

  Greater_Op, LessThan_Op, Greater_Equal_Op,
  LessThan_Equal_Op(TypeIs_Scalar, TypeIs_Scalar): TypeIs_int;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

Operators[25]:

```
INDICATION
  Equal_Equal_Indication: Equality_Op,  Ptr_Eq_Op;
  Bang_Equal_Indication:  Not_Equal_Op, Ptr_NEq_Op;

OPER
  Equality_Op, Not_Equal_Op(TypeIs_Scalar, TypeIs_Scalar): TypeIs_int;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

Operators[26]:

```
INDICATION
  Amper_Indication:       Bitwise_And_Op;
  Caret_Indication:       Bitwise_XOr_Op;
  Bar_Indication:         Bitwise_Or_Op;
  Amper_Amper_Indication: Logical_And_Op;
  Bar_Bar_Indication:     Logical_Or_Op;
  Conditional_Indication: Arith_Cond_Op, Void_Cond_Op, Ptr_Cond_Op;

SET TypeIs_RHS_Scalar = TypeIs_Scalar;

OPER
  Bitwise_And_Op, Bitwise_XOr_Op,
  Bitwise_Or_Op(TypeIs_Integral, TypeIs_Integral): TypeIs_Integral;
```

```
  Logical_And_Op,
  Logical_Or_Op(TypeIs_Scalar, TypeIs_RHS_Scalar): TypeIs_int;
  Arith_Cond_Op(TypeIs_Scalar,TypeIs_Arithmetic,TypeIs_Arithmetic): TypeIs_Arithmetic;
  Void_Cond_Op (TypeIs_Scalar,TypeIs_Void,TypeIs_Void): TypeIs_Void;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

FIXME: The standard allows balancing of two compatible struct or union types. The best way to handle this is probably to have a common type that all such types can be coerced to, followed by more stringent testing.

Operators[27]:

```
INDICATION
   Equal_Indication:
     Assign_Op, Ptr_Assign_Op, Enum_Assign_Op,
     Struct_Assign_Op, Union_Assign_Op, Ptr_Void_Assign_Op,
     Ptr_Void_Void_Assign_Op;

SET TypeIs_RHS_Arithmetic = TypeIs_Arithmetic;

OPER
  Assign_Op(TypeIs_Arithmetic, TypeIs_RHS_Arithmetic): TypeIs_Arithmetic;
  Ptr_Void_Void_Assign_Op
    (TypeIs_VoidPointer, TypeIs_VoidPointer): TypeIs_VoidPointer;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

Operators[28]:

```
INDICATION
  Star_Equal_Indication:            Mult_Eq_Op;
  Slash_Equal_Indication:           Div_Eq_Op;
  Percent_Equal_Indication:         Mod_Eq_Op;
  Plus_Equal_Indication:            Plus_Eq_Op,  Ptr_Plus_Eq_Op;
  Minus_Equal_Indication:           Minus_Eq_Op, Ptr_Minus_Eq_Op;
  Less_Less_Equal_Indication:       Bitwise_Shift_Left_Eq_Op;
  Greater_Greater_Equal_Indication: Bitwise_Shift_Right_Eq_Op;
  Amper_Equal_Indication:           Bitwise_And_Eq_Op;
  Caret_Equal_Indication:           Bitwise_XOr_Eq_Op;
  Bar_Equal_Indication:             Bitwise_Or_Eq_Op;

SET TypeIs_RHS_Integral = TypeIs_Integral;

OPER
  Mult_Eq_Op, Div_Eq_Op, Plus_Eq_Op,
  Minus_Eq_Op( TypeIs_Arithmetic, TypeIs_RHS_Arithmetic ) : TypeIs_Arithmetic;
  Mod_Eq_Op, Bitwise_Shift_Left_Eq_Op, Bitwise_Shift_Right_Eq_Op,
  Bitwise_And_Eq_Op, Bitwise_XOr_Eq_Op,
  Bitwise_Or_Eq_Op( TypeIs_Integral, TypeIs_RHS_Integral ) : TypeIs_Integral;
```

This macro is defined in definitions 19, 20, 21, 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 4.

4.2 Associating types with identifiers

Each type is represented by a definition table key, and these keys are related to the abstract syntax tree by the Eli `Typing` module:

Instantiate modules[29]:

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

This macro is defined in definitions 29, 49, 50, and 51.
This macro is invoked in definition 3.

This module classifies identifiers as either "typed" (representing entities, such as variables, that have a type property) or "type definition" (representing types themselves). Each occurrence of the identifier is either a "definition", at which the type property is set, or a "use" at which it is made available.

Certain constructs are also classified as "type denotations". These are the constructs that build new types from existing types.

Associate types with identifiers[30]:

```
SYMBOL Declaration          INHERITS TypedDefinition               END;
SYMBOL Declarator           INHERITS TypedDefinition               END;

SYMBOL TagDef               INHERITS TypeDefDefId, ChkTypeDefDefId END;
SYMBOL TagUse               INHERITS TypeDefUseId, ChkTypeDefUseId END;
SYMBOL ForwardDef           INHERITS TypeDefDefId, ChkTypeDefDefId END;
SYMBOL ForwardUse           INHERITS TypeDefDefId, ChkTypeDefDefId END;

SYMBOL MemberIdDef          INHERITS TypedDefId                    END;

SYMBOL enumeration_constant INHERITS TypedDefId                   END;
SYMBOL IdDef                INHERITS TypedDefId                    END;
SYMBOL IdUse                INHERITS TypedUseId,   ChkTypedUseId   END;

SYMBOL TypeIdDef            INHERITS TypeDefDefId, ChkTypeDefDefId END;
SYMBOL TypeIdUse            INHERITS TypeDefUseId, ChkTypeDefUseId END;
```

This macro is defined in definitions 30, 31, 32, and 33.
This macro is invoked in definition 1.

Definition table keys for derived types are created at tree nodes inheriting the `TypeDenotation` role of the `Typing` module, and passed to nodes inheriting other roles via attribute computations. Structures, unions and enumerations are specifiers; pointers, arrays, and functions are declarators.

Associate types with identifiers[31]:

```
ATTR IsStruct: int;
```

```
RULE: struct_or_union ::= 'struct' COMPUTE
  struct_or_union.IsStruct=1;
END;

RULE: struct_or_union ::= 'union' COMPUTE
  struct_or_union.IsStruct=0;
END;

SYMBOL struct_declaration_list INHERITS TypeDenotation, OperatorDefs COMPUTE
  SYNT.Pointer=NewType();
  SYNT.GotType=
    AddTypeToBlock(
      THIS.Pointer,
      PointerTypes,
      SingleDefTableKeyList(THIS.Type));
  SYNT.GotOper=
    ORDER(
      IF(THIS.IsStruct,
        InstClass0(TypeIs_Struct,THIS.Type),
        InstClass0(TypeIs_Union, THIS.Type)),
      InstClass1(TypeIs_Pointer,THIS.Pointer,THIS.Type));
END;

RULE: struct_or_union_specifier ::=
        struct_or_union TagDef '{' struct_declaration_list '}' COMPUTE
  struct_declaration_list.IsStruct=struct_or_union.IsStruct;
  struct_or_union_specifier.Type=struct_declaration_list.Type;
  TagDef.Type=struct_declaration_list.Type;
END;

RULE: struct_or_union_specifier ::=
        struct_or_union '{' struct_declaration_list '}' COMPUTE
  struct_declaration_list.IsStruct=struct_or_union.IsStruct;
  struct_or_union_specifier.Type=struct_declaration_list.Type;
END;

RULE: struct_or_union_specifier ::= struct_or_union ForwardUse COMPUTE
  struct_or_union_specifier.Type=ForwardUse.Type;
END;

SYMBOL enumerator_list INHERITS TypeDenotation, OperatorDefs COMPUTE
  SYNT.Pointer=NewType();
  SYNT.GotType=
    AddTypeToBlock(
      THIS.Pointer,
      PointerTypes,
      SingleDefTableKeyList(THIS.Type));
  SYNT.GotOper=
    ORDER(
      InstClass0(TypeIs_Enum,THIS.Type),
```

```
      InstClass1(TypeIs_Pointer,THIS.Pointer,THIS.Type));
END;

RULE: enum_specifier ::= 'enum' TagDef '{' enumerator_list '}' COMPUTE
  enum_specifier.Type=enumerator_list.Type;
  TagDef.Type=enumerator_list.Type;
END;

RULE: enum_specifier ::= 'enum' '{' enumerator_list '}' COMPUTE
  enum_specifier.Type=enumerator_list.Type;
END;

RULE: enum_specifier ::= 'enum' TagUse COMPUTE
  enum_specifier.Type=TagUse.Type;
END;
```

This macro is defined in definitions 30, 31, 32, and 33.
This macro is invoked in definition 1.

A type denoted by a specifier is independent of other types. Types denoted by declarator nodes, on the other hand, create a type by modifying another type. In each of these constructs, the original type comes from a set of specifiers. That original type is then passed through a nest of declarators, each of which creates a new type based on the type it receives.

Associate types with identifiers[32]:

```
CLASS SYMBOL Declaration COMPUTE
  SYNT.Type=NoKey;
END;

TREE SYMBOL declaration INHERITS Declaration END;

RULE: declaration ::= Specifiers init_declarator_list_opt ';' COMPUTE
  declaration.Type=Specifiers.Type;
END;

TREE SYMBOL struct_declaration INHERITS Declaration END;

RULE: struct_declaration ::= Specifiers struct_declarator_list ';' COMPUTE
  struct_declaration.Type=Specifiers.Type;
END;

TREE SYMBOL function_definition INHERITS Declaration END;

RULE: function_definition ::=
        Specifiers declaration_list FunctionDecl function_body COMPUTE
  function_definition.Type=Specifiers.Type;
END;

TREE SYMBOL parameter_declaration INHERITS Declaration END;
```

```
RULE: parameter_declaration ::= Specifiers COMPUTE
  parameter_declaration.Type=Specifiers.Type;
END;

RULE: parameter_declaration ::= Specifiers ParameterDecl COMPUTE
  parameter_declaration.Type=Specifiers.Type;
END;

RULE: parameter_declaration ::= Specifiers abstract_declarator COMPUTE
  parameter_declaration.Type=Specifiers.Type;
END;

TREE SYMBOL par_declaration INHERITS Declaration END;

RULE: par_declaration ::= Specifiers par_id_decls ';' COMPUTE
  par_declaration.Type=Specifiers.Type;
END;

TREE SYMBOL type_name INHERITS Declaration END;

RULE: type_name ::= Specifiers COMPUTE
  type_name.Type=Specifiers.Type;
END;

RULE: type_name ::= Specifiers abstract_declarator COMPUTE
  type_name.Type=Specifiers.Type;
END;
```

This macro is defined in definitions 30, 31, 32, and 33.
This macro is invoked in definition 1.

`ForwardDef` and `TagUse` require special treatment. According to Section 6.5.2.3 of the standard, each may define a new incomplete type. If the type is to be completed, a subsequent declaration can be given to specify the content. The strategy is to specify the type as the incomplete type `void` if and only if a type has not already been specified. When a type has already been specified, specify that type again. Since a construct in which the tag is followed by a bracketed list unconditionally set the type, that will override any type set in any other context.

Associate types with identifiers[33]:

```
SYMBOL ForwardType COMPUTE SYNT.Type=GetDefer(THIS.Key,TypeIs_void); END;
SYMBOL ForwardDef  INHERITS ForwardType END;
SYMBOL ForwardUse  INHERITS ForwardType END;
```

This macro is defined in definitions 30, 31, 32, and 33.
This macro is invoked in definition 1.

4.2.1 Analyze specifiers

In order to attach an error report to the first specifier that cannot be accepted, specifier lists are analyzed left-to-right using a finite-state machine. Each specifier is an input to the machine, and the state of the machine is a component of the value of a chain passing through the specifiers.

Declaration specifiers[34]:

```
ATTR  InitialType: DefTableKey;
CHAIN Specification: SpecData;

SYMBOL Specifiers COMPUTE
  CHAINSTART HEAD.Specification=InitSpecifiers(THIS.InitialType);
  INH.InitialType=NoKey;
  SYNT.Type=CompleteType(TAIL.Specification);
END;
```

This macro is defined in definitions 34 and 40.
This macro is invoked in definition 1.

The transition table for the finite-state machine is:

Finite-state machine[35]:

```
/*                          u
 *                          n
 *              d         s  s
 *              o  s      i  i
 *           c  u  h  l   g  g
 *        i  h  b  o  o   n  n
 *        n  a  l  r  n   e  e
 *        t  r  e  t  g   d  d
 ***/
/* 0*/ {{ 0, 1, 9, 4, 7,11, 6}, TypeIs_int},
/* 1*/ {{ 1, 1, 1, 1, 1, 2, 3}, TypeIs_char},
/* 2*/ {{ 2, 2, 2, 2, 2, 2, 2}, TypeIs_signed_char},
/* 3*/ {{ 3, 3, 3, 3, 3, 3, 3}, TypeIs_unsigned_char},
/* 4*/ {{ 4, 4, 4, 4, 4, 4, 5}, TypeIs_short},
/* 5*/ {{ 5, 5, 5, 5, 5, 5, 5}, TypeIs_unsigned_short},
/* 6*/ {{ 6, 3, 6, 5, 8, 6, 6}, TypeIs_unsigned_int},
/* 7*/ {{ 7, 7, 7, 7, 7, 7, 8}, TypeIs_long},
/* 8*/ {{ 8, 8, 8, 8, 8, 8, 8}, TypeIs_unsigned_long},
/* 9*/ {{ 9, 9, 9, 9,10, 9, 9}, TypeIs_double},
/*10*/ {{10,10,10,10,10,10,10}, TypeIs_long_double},
/*11*/ {{11, 2,11, 4, 7,11,11}, TypeIs_int}
```

This macro is invoked in definition 55.

`Specification` is the chain carrying state and other information from one specifier to the next. It is a structure of type `SpecData`:

Specification data[36]:

```
typedef struct {
  int CurrentState;          /* The current state of the machine */
  long KeywordSet;           /* The specifiers seen so far       */
  DefTableKey SpecifiedType; /* A type specified explicitly      */
} SpecData;
```

This macro is invoked in definition 54.

InitSpecifiers(DefTableKey key)[37]:

```
{ SpecData CurrentData;
  CurrentData.CurrentState  = 0;
  CurrentData.KeywordSet    = 0;
  CurrentData.SpecifiedType = key;
  return CurrentData; }
```

This macro is invoked in definition 55.

Two operations are applied to `Specification` at each specifier. The first decides whether that specifier is legal, given its left context as embodied in `Specification`, and the second determines the new value of `Specification` needed to embody the left context with the current specifier added. These operations and the state table above attempt to minimize cascading errors by behaving as though an erroneous specifier was omitted.

NextSpecifier(TypeSpecifier kw, SpecData chain)[38]:

```
/* Decide whether this keyword is legal in this context
 *   On entry-
 *     Exclude[kw]=Bit vector specifying the set of keywords that cannot
 *                 occur in combination with kw
 *     chain defines the current left context
 *   On exit-
 *     NextSpecifier=1 if this keyword is legal in the context of chain
 *                   0 otherwise
 ***/
{ return (Exclude[kw] & chain.KeywordSet) == 0; }
```

This macro is invoked in definition 55.

UpdateSpecification(DefTableKey type, TypeSpecifier kw, SpecData chain)[39]:

```
/* Update the context on the basis of a specifier
 *   On entry-
 *     type=Explicit type if determined by this specifier
 *          NoKey otherwise
 *     kw=Enumerated constant defining this keyword
 *     chain defines the current left context
 *   On exit-
 *     UpdateSpecification defines the left context including this keyword
 ***/
{ if ((Exclude[kw] & chain.KeywordSet) == 0) {
    if (type != NoKey) chain.SpecifiedType = type;
    chain.KeywordSet |= (1 << kw);
    chain.CurrentState = State[chain.CurrentState].Next[FSMInput[kw]];
  }
  return chain;
}
```

This macro is invoked in definition 55.

A computation at each specifier node invokes these functions with appropriate arguments. Many of the keywords have the same type of computation, differing only in one or more of the parameters:

Declaration specifiers[40]:

Simple declaration specifier[41](‘`typedef`’)
Simple declaration specifier[41](‘`extern`’)
Simple declaration specifier[41](‘`static`’)
Simple declaration specifier[41](‘`auto`’)
Simple declaration specifier[41](‘`register`’)

Specifier that determines type[42](‘`void`’,‘`TypeIs_void`’)
Simple declaration specifier[41](‘`char`’)
Simple declaration specifier[41](‘`short`’)
Simple declaration specifier[41](‘`int`’)
Simple declaration specifier[41](‘`long`’)
Specifier that determines type[42](‘`float`’,‘`TypeIs_float`’)
Simple declaration specifier[41](‘`double`’)
Simple declaration specifier[41](‘`signed`’)
Simple declaration specifier[41](‘`unsigned`’)
Non-keyword specifier[43](‘`struct_or_union_specifier`’)
Non-keyword specifier[43](‘`enum_specifier`’)
Non-keyword specifier[43](‘`TypeIdUse`’)

Simple declaration specifier[41](‘`const`’)
Simple declaration specifier[41](‘`volatile`’)

This macro is defined in definitions 34 and 40.
This macro is invoked in definition 1.

Simple declaration specifier[41](◇1):

```
RULE: Specifier ::= '◇1' COMPUTE
  Specifier.ok=
    NextSpecifier(Kwd_◇1,Specifier.Specification);
  Specifier.Specification=
    UpdateSpecification(NoKey,Kwd_◇1,Specifier.Specification);
END;
```

This macro is invoked in definition 40.

Specifier that determines type[42](◇2):

```
RULE: Specifier ::= '◇1' COMPUTE
  Specifier.ok=
    NextSpecifier(Kwd_typeid,Specifier.Specification);
  Specifier.Specification=
    UpdateSpecification(◇2,Kwd_typeid,Specifier.Specification);
END;
```

This macro is invoked in definition 40.

Non-keyword specifier[43](◇1):

```
RULE: Specifier ::= ◇1 COMPUTE
  Specifier.ok=
    NextSpecifier(Kwd_typeid,Specifier.Specification);
  Specifier.Specification=
    UpdateSpecification(◇1.Type,Kwd_typeid,Specifier.Specification);
END;
```

This macro is invoked in definition 40.

After all specifiers have been seen, the type is determined either by an explicit type given by one of them or by the current state of the machine. The effect of any type qualifiers must also be taken into account.

CompleteType(SpecData chain)[44]:

```
/* Determine the specified type
 *   On entry-
 *     chain defines the current left context
 *   On exit-
 *     CompleteType=type specified by the sequence
 ***/
{ DefTableKey type;

  type = chain.SpecifiedType != NoKey ? chain.SpecifiedType
                                      : State[chain.CurrentState].Type;

  /* FIXME
  if (InSpecifierSet(Kwd_const, chain)) {
    if (InSpecifierSet(Kwd_volatile, chain)) return KeyForConstVolatile(type);
    return KeyForConst(type);
  }
  if (InSpecifierSet(Kwd_volatile, chain)) return KeyForVolatile(type);
  */
  return type;
}
```

This macro is invoked in definition 55.

The relationships among qualified and unqualified types are explained in Section 4.2.2.
Here is a list of the specifier and qualifier keywords:

Specifier keywords[45]:

```
KWD(Kwd_typedef,  0, (ClassBits)),
KWD(Kwd_extern,   0, (ClassBits)),
KWD(Kwd_static,   0, (ClassBits)),
KWD(Kwd_auto,     0, (ClassBits)),
KWD(Kwd_register, 0, (ClassBits)),

/*      void    is represented by Kwd_typeid  */
KWD(Kwd_char,     1, ((1<<Kwd_typeid) | TypeBits | SizeBits)),
KWD(Kwd_short,    3, ((1<<Kwd_typeid) | SizeBits | (1<<Kwd_char))),
```

```
KWD(Kwd_int,      0, ((1<<Kwd_typeid) | TypeBits)),
KWD(Kwd_long,     4, ((1<<Kwd_typeid) | SizeBits | (1<<Kwd_char))),
/*      float   is represented by Kwd_typeid */
KWD(Kwd_double,   2, ((1<<Kwd_typeid) | TypeBits | (1<<Kwd_short) |
SignBits)),
KWD(Kwd_signed,   5, ((1<<Kwd_typeid) | SignBits)),
KWD(Kwd_unsigned, 6, ((1<<Kwd_typeid) | SignBits)),

KWD(Kwd_typeid,   0, ((1<<Kwd_typeid) - (1<<Kwd_char))),

KWD(Kwd_const,    0, ((1<<Kwd_const) | (1<<Kwd_volatile))),
KWD(Kwd_volatile, 0, ((1<<Kwd_const) | (1<<Kwd_volatile)))
```

This macro is invoked in definitions 54 and 55.

The KWD macro is used simply as a documentation aid, pulling together several aspects of the specifiers. The first argument is the enumerated constant used to represent the declaration specifier internally. The second argument is that specifier's input value for the finite state machine that ultimately determines the type represented by the sequence of specifiers. Finally, the third argument gives the set of declaration specifiers that are incompatible with the specifier represented by the keyword. Abbreviations, defined as follows, are used for specific groups of bits:

Abbreviations for sets of bits[46]:

```
#define ClassBits ((1<<(Kwd_register + 1)) - (1<<Kwd_typedef))
#define TypeBits  ((1<<Kwd_char)   | (1<<Kwd_int)      | (1<<Kwd_double))
#define SizeBits  ((1<<Kwd_short)  | (1<<Kwd_long))
#define SignBits  ((1<<Kwd_signed) | (1<<Kwd_unsigned))
```

This macro is invoked in definition 54.

4.2.2 Analyze declarators

New types are created in declarator contexts. In each case, the new type is related to an existing type that is obtained from the parent of the context.

Relationships among types[47]:

```
ATTR ExistingType, TotalType: DefTableKey;

CLASS SYMBOL Declarator INHERITS TypeDenotation, OperatorDefs COMPUTE
  INH.ExistingType=INCLUDING (Declarator.Type, Declaration.Type);
END;

CLASS SYMBOL DeclaredId COMPUTE
  SYNT.TotalType=INCLUDING (Declarator.Type, Declaration.Type);
END;

TREE SYMBOL MemberIdDef INHERITS DeclaredId END;
TREE SYMBOL IdDef       INHERITS DeclaredId END;
```

```
CLASS SYMBOL PointerDeclarator INHERITS Declarator COMPUTE
  SYNT.GotType=
    AddTypeToBlock(
      THIS.Type,
      PointerTypes,     /* FIXME: Pointer types have extra qualifiers */
      SingleDefTableKeyList(THIS.ExistingType));
  SYNT.GotOper=InstClass1(TypeIs_Pointer,THIS.Type,THIS.ExistingType);
END;

TREE SYMBOL pointer_declarator           INHERITS PointerDeclarator END;
TREE SYMBOL member_pointer_declarator    INHERITS PointerDeclarator END;
TREE SYMBOL pointer_abstract_declarator  INHERITS PointerDeclarator END;

ATTR Pointer: DefTableKey;

CLASS SYMBOL ArrayDeclarator INHERITS Declarator COMPUTE
  SYNT.Pointer=NewType();
  SYNT.GotType=
    ORDER(
      AddTypeToBlock(
        THIS.Pointer,
        PointerTypes,   /* FIXME: This doesn't account for array size */
        SingleDefTableKeyList(THIS.ExistingType)),
      AddTypeToBlock(
        THIS.Type,
        ArrayTypes,     /* FIXME: This doesn't account for array size */
        SingleDefTableKeyList(THIS.ExistingType)));
  SYNT.GotOper=
    InstClass2(TypeIs_Array,THIS.Type,THIS.ExistingType,THIS.Pointer);
END;

TREE SYMBOL array_declarator             INHERITS ArrayDeclarator END;
TREE SYMBOL member_array_declarator      INHERITS ArrayDeclarator END;
TREE SYMBOL array_abstract_declarator    INHERITS ArrayDeclarator END;

CLASS SYMBOL FunctionDeclarator INHERITS Declarator COMPUTE
  SYNT.Pointer=NewType();
  SYNT.GotType=
    ORDER(
      AddTypeToBlock(
        THIS.Pointer,
        PointerTypes,
        SingleDefTableKeyList(THIS.Type)),
      AddTypeToBlock(
        THIS.Type,
        FunctionTypes,
        ConsDefTableKeyList(
          THIS.ExistingType,
          CONSTITUENT OpndTypeListRoot.OpndTypeList
            SHIELD (Declarator, OpndTypeListRoot))));
END;
```

```
TREE SYMBOL parameter_type_list INHERITS OpndTypeListRoot END;
TREE SYMBOL parameters          INHERITS OpndTypeListRoot END;
TREE SYMBOL ParameterType       INHERITS OpndTypeListElem END;
TREE SYMBOL parameter_id        INHERITS OpndTypeListElem END;
TREE SYMBOL DotDotDot           INHERITS OpndTypeListElem END;
TREE SYMBOL parameter_id        INHERITS TypeDefUseId, ChkTypeDefUseId END;
TREE SYMBOL ParameterTypeId     INHERITS TypeDefDefId                  END;

RULE: DotDotDot ::= '...' COMPUTE
  DotDotDot.Type=VarArgType;
END;

RULE: ParameterTypeId ::= ParameterDecl COMPUTE
  ParameterTypeId.Key=ParameterDecl.Key;
  ParameterTypeId.Type=ParameterDecl.TotalType;
END;

RULE: ParameterType ::= parameter_declaration COMPUTE
  ParameterType.Type=parameter_declaration.TotalType;
END;

TREE SYMBOL function_declarator INHERITS FunctionDeclarator END;

RULE: function_declarator ::= declarator '(' parameter_type_list ')' COMPUTE
  function_declarator.GotOper=
    ORDER(
      InstClass1(
        TypeIs_Function,
        function_declarator.Type,
        function_declarator.Pointer),
      ListOperator(
        function_declarator.Type,
        NoOprName,
        parameter_type_list.OpndTypeList,
        function_declarator.ExistingType));
END;

RULE: function_declarator ::= declarator '(' parameters ')' COMPUTE
  function_declarator.GotOper=
    ORDER(
      InstClass1(
        TypeIs_Function,
        function_declarator.Type,
        function_declarator.Pointer),
      ListOperator(
        function_declarator.Type,
        NoOprName,
        parameters.OpndTypeList,
        function_declarator.ExistingType));
END;
```

```
TREE SYMBOL function_abstract_declarator INHERITS FunctionDeclarator END;

RULE: function_abstract_declarator ::= '(' parameter_type_list ')' COMPUTE
  function_abstract_declarator.GotOper=
    ORDER(
      InstClass1(
        TypeIs_Function,
        function_abstract_declarator.Type,
        function_abstract_declarator.Pointer),
      ListOperator(
        function_abstract_declarator.Type,
        NoOprName,
        parameter_type_list.OpndTypeList,
        function_abstract_declarator.ExistingType));
END;

RULE: function_abstract_declarator ::=
                   abstract_declarator '(' parameter_type_list ')' COMPUTE
  function_abstract_declarator.GotOper=
    ORDER(
      InstClass1(
        TypeIs_Function,
        function_abstract_declarator.Type,
        function_abstract_declarator.Pointer),
      ListOperator(
        function_abstract_declarator.Type,
        NoOprName,
        parameter_type_list.OpndTypeList,
        function_abstract_declarator.ExistingType));
END;

TREE SYMBOL member_function_declarator INHERITS FunctionDeclarator END;

RULE: member_function_declarator ::=
        member_declarator '(' parameter_type_list ')' COMPUTE
  member_function_declarator.GotOper=
    ORDER(
      InstClass1(
        TypeIs_Function,
        member_function_declarator.Type,
        member_function_declarator.Pointer),
      ListOperator(
        member_function_declarator.Type,
        NoOprName,
        parameter_type_list.OpndTypeList,
        member_function_declarator.ExistingType));
END;

RULE: declarator ::= TypeIdDef COMPUTE
  TypeIdDef.Type=INCLUDING (Declarator.Type, Declaration.Type);
```

```
END;

RULE: type_name ::= Specifiers COMPUTE
  type_name.TotalType=Specifiers.Type;
END;

RULE: type_name ::= Specifiers abstract_declarator COMPUTE
  type_name.TotalType=abstract_declarator.TotalType;
END;

RULE: abstract_declarator ::= pointer_abstract_declarator COMPUTE
  abstract_declarator.TotalType=pointer_abstract_declarator.TotalType;
END;

RULE: abstract_declarator ::= array_abstract_declarator COMPUTE
  abstract_declarator.TotalType=array_abstract_declarator.TotalType;
END;

RULE: abstract_declarator ::= function_abstract_declarator COMPUTE
  abstract_declarator.TotalType=function_abstract_declarator.TotalType;
END;

RULE: pointer_abstract_declarator ::= '*' Specifiers COMPUTE
  pointer_abstract_declarator.TotalType=pointer_abstract_declarator.Type;
END;

RULE: pointer_abstract_declarator ::=
        '*' Specifiers abstract_declarator COMPUTE
  pointer_abstract_declarator.TotalType=abstract_declarator.TotalType;
END;

RULE: array_abstract_declarator ::= '[' constant_expression ']' COMPUTE
        array_abstract_declarator.TotalType=array_abstract_declarator.Type;
END;

RULE: array_abstract_declarator ::=
        abstract_declarator '[' constant_expression ']' COMPUTE
  array_abstract_declarator.TotalType=abstract_declarator.TotalType;
END;

RULE: function_abstract_declarator ::= '(' parameter_type_list ')' COMPUTE
  function_abstract_declarator.TotalType=function_abstract_declarator.Type;
END;

RULE: function_abstract_declarator ::=
        abstract_declarator '(' parameter_type_list ')' COMPUTE
  function_abstract_declarator.TotalType=abstract_declarator.TotalType;
END;

RULE: declarator ::= TypeIdDef COMPUTE
  declarator.TotalType=TypeIdDef.Type;
```

```
END;

RULE: declarator ::= IdDef COMPUTE
  declarator.TotalType=IdDef.Type;
END;

RULE: declarator ::= pointer_declarator COMPUTE
  declarator.TotalType=pointer_declarator.TotalType;
END;

RULE: declarator ::= array_declarator COMPUTE
  declarator.TotalType=array_declarator.TotalType;
END;

RULE: declarator ::= function_declarator COMPUTE
  declarator.TotalType=function_declarator.TotalType;
END;

RULE: pointer_declarator ::= '*' Specifiers declarator COMPUTE
  pointer_declarator.TotalType=declarator.TotalType;
END;

RULE: array_declarator ::= declarator '[' constant_expression ']' COMPUTE
  array_declarator.TotalType=declarator.TotalType;
END;

RULE: function_declarator ::= declarator '(' parameter_type_list ')' COMPUTE
  function_declarator.TotalType=declarator.TotalType;
END;

RULE: function_declarator ::= declarator '(' parameters ')' COMPUTE
  function_declarator.TotalType=declarator.TotalType;
END;

RULE: parameter_declaration ::= Specifiers ParameterDecl COMPUTE
  parameter_declaration.TotalType=ParameterDecl.TotalType;
END;

RULE: parameter_declaration ::= Specifiers abstract_declarator COMPUTE
  parameter_declaration.TotalType=abstract_declarator.TotalType;
END;

RULE: parameter_declaration ::= Specifiers COMPUTE
  parameter_declaration.TotalType=Specifiers.Type;
END;

RULE: ParameterDecl ::= declarator COMPUTE
  ParameterDecl.TotalType=declarator.TotalType;
END;
```

This macro is defined in definitions 47.

This macro is invoked in definition 1.

Define keys, properties, and access functions[48]:

```
PointerTypes;
ArrayTypes;
FunctionTypes;
```

This macro is defined in definitions 48.
This macro is invoked in definition 2.

Instantiate modules[49]:

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

This macro is defined in definitions 29, 49, 50, and 51.
This macro is invoked in definition 3.

4.3 Determining the types yielded by expressions

Operator identification (also called *overload resolution*) is the process of determining the type of value yielded by each expression in the program.

Instantiate modules[50]:

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

This macro is defined in definitions 29, 49, 50, and 51.
This macro is invoked in definition 3.

This process depends on the type system of the language.

The formal description of the type system given in Section 4.1 results in a database of relationships among types and operators that can be used by a standard set of computational roles to implement operator identification.

Each of the basic type identifiers appearing in Section 4.1.1 denotes a definition table key that has the `OilType` property. The value of the `OilType` property is the corresponding type in the operator identification database. Section 4.1.1 defines only one database type for a C derived type: `void*`. All other C derived types are considered program-dependent and must be added to the database by computations over the abstract syntax tree. These computations are discussed in Section **??**.

4.3.1 Operator indications

A correspondence must be set up between the C operator characters and the operator indications used in the formal specification (Section 4.1.3).

Instantiate modules[51]:

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

This macro is defined in definitions 29, 49, 50, and 51.
This macro is invoked in definition 3.

Operator.d[52]:

```
PreDefInd('++',    post_lvalue_opr,     Plus_Plus_Indication)
PreDefInd('--',    post_lvalue_opr,     Minus_Minus_Indication)

PreDefInd('++',    lvalue_operator,     Plus_Plus_Indication)
PreDefInd('--',    lvalue_operator,     Minus_Minus_Indication)
PreDefInd('sizeof',lvalue_operator,     Sizeof_Indication)

PreDefInd('=',     assignment_operator, Equal_Indication)
PreDefInd('*=',    assignment_operator, Star_Equal_Indication)
PreDefInd('/=',    assignment_operator, Slash_Equal_Indication)
PreDefInd('%=',    assignment_operator, Percent_Equal_Indication)
PreDefInd('+=',    assignment_operator, Plus_Equal_Indication)
PreDefInd('-=',    assignment_operator, Minus_Equal_Indication)
PreDefInd('<<=',   assignment_operator, Less_Less_Equal_Indication)
PreDefInd('>>=',   assignment_operator, Greater_Greater_Equal_Indication)
PreDefInd('&=',    assignment_operator, Amper_Equal_Indication)
PreDefInd('^=',    assignment_operator, Caret_Equal_Indication)
PreDefInd('|=',    assignment_operator, Bar_Equal_Indication)

PreDefInd('~',     normal_operator,     Tilde_Indication)
PreDefInd('!',     normal_operator,     Bang_Indication)

PreDefInd('*',     normal_operator,     Star_Indication)
PreDefInd('/',     normal_operator,     Slash_Indication)
PreDefInd('%',     normal_operator,     Percent_Indication)

PreDefInd('+',     normal_operator,     Plus_Indication)
PreDefInd('-',     normal_operator,     Minus_Indication)

PreDefInd('<<',    normal_operator,     Less_Less_Indication)
PreDefInd('>>',    normal_operator,     Greater_Greater_Indication)

PreDefInd('<',     normal_operator,     Less_Indication)
PreDefInd('>',     normal_operator,     Greater_Indication)
PreDefInd('<=',    normal_operator,     Less_Equal_Indication)
PreDefInd('>=',    normal_operator,     Greater_Equal_Indication)

PreDefInd('==',    normal_operator,     Equal_Equal_Indication)
PreDefInd('!=',    normal_operator,     Bang_Equal_Indication)

PreDefInd('&',     normal_operator,     Amper_Indication)

PreDefInd('^',     normal_operator,     Caret_Indication)

PreDefInd('|',     normal_operator,     Bar_Indication)

PreDefInd('&&',    logical_operator,    Amper_Amper_Indication)
PreDefInd('||',    logical_operator,    Bar_Bar_Indication)
```

This macro is attached to a non-product file.

4.3.2 Expression contexts

Certain abstract tree node contexts inherit computational roles from the operator identification module.

Operator identification[53]:

```
SYMBOL post_lvalue_opr     INHERITS OperatorSymbol   END;
SYMBOL lvalue_operator     INHERITS OperatorSymbol   END;
SYMBOL assignment_operator INHERITS OperatorSymbol   END;
SYMBOL normal_operator     INHERITS OperatorSymbol   END;
SYMBOL logical_operator    INHERITS OperatorSymbol   END;

SYMBOL Expression          INHERITS ExpressionSymbol END;
SYMBOL RHSExpr             INHERITS ExpressionSymbol END;
SYMBOL DerefExpr           INHERITS ExpressionSymbol END;

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

RULE: Expression ::= character_constant COMPUTE
  PrimaryContext(Expression,TypeIs_char);
END;

RULE: Expression ::= floating_constant COMPUTE
  PrimaryContext(Expression,TypeIs_double);
END;

RULE: Expression ::= integer_constant COMPUTE
/* FIXME: Check the suffix */
  PrimaryContext(
    Expression,
    IF(EQ(strtol(StringTable(integer_constant),0,0),0),
      TypeIs_NULL,
      TypeIs_int));
END;
RULE: Expression ::= StringSeq
COMPUTE
  PrimaryContext(Expression,TypeIs_string);
END;

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

SYMBOL Arguments INHERITS OpndExprListRoot COMPUTE
  SYNT.LstMsg=
    IF(THIS.LstErr,
      IF(AND(EQ(THIS.LstTyp,TypeIs_void),NOT(THIS.LstNxt)),
        "A single 'void' means no arguments, so no message",
```

```
        message(ERROR,"Too few arguments",0,COORDREF)));
END;

SYMBOL Argument  INHERITS OpndExprListElem END;

RULE: Argument ::= Expression COMPUTE
  ConversionContext(Argument,,Expression);
  Indication(Cast_Indication);
END;

RULE: Expression ::= Expression '(' Arguments ')' COMPUTE
  ListContext(Expression[1],,Arguments);
  Indication(Expression[2].Type);
END;

SYMBOL MemberIdUse INHERITS TypedUseId END;

RULE: Expression ::= Expression '.' MemberIdUse
COMPUTE
  MemberIdUse.MemberScopeKey=Expression[2].Type;
  PrimaryContext(Expression[1],MemberIdUse.Type);
END;

RULE: Expression ::= DerefExpr '->' MemberIdUse COMPUTE
  MemberIdUse.MemberScopeKey=DerefExpr.Type;
  PrimaryContext(Expression,MemberIdUse.Type);
END;

RULE: DerefExpr ::= Expression COMPUTE
  MonadicContext(DerefExpr,,Expression);
  Indication(Star_Indication);
END;

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

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

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

RULE: Expression ::= 'sizeof' '(' type_name ')' COMPUTE
  PrimaryContext(Expression[1],TypeIs_int);
END;

RULE: Expression ::= '(' type_name ')' Expression COMPUTE
  PrimaryContext(Expression[1],type_name.TotalType);
```

```
  RootContext(type_name.TotalType,,Expression[2]);
  Indication(Cast_Indication);
END;

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

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

RULE: Expression ::= Expression '?' Expression ':' Expression COMPUTE
  BalanceContext(Expression[1],Expression[3],Expression[4]);
  Expression[2].Required=scalarType;
END;

RULE: Expression ::= Expression assignment_operator RHSExpr COMPUTE
  DyadicContext(Expression[1],assignment_operator,Expression[2],RHSExpr);
END;

RULE: RHSExpr ::= Expression COMPUTE
  ConversionContext(RHSExpr,,Expression);
  Indication(Cast_Indication);
END;

RULE: Expression ::= Expression ',' Expression COMPUTE
  TransferContext(Expression[1],Expression[3]);
END;
```

This macro is defined in definitions 53.
This macro is invoked in definition 1.

4.4 Support code

Type.h[54]:

```
#ifndef SEMANTIC_H
#define SEMANTIC_H

#include "eliproto.h"
#include "deftbl.h"
#include "oiladt2.h"
#include "Strings.h"
```

Abbreviations for sets of bits[46]

```
#define KWD(w,i,s) w
typedef enum {
```

Specifier keywords[45]

```
} TypeSpecifier;
#undef KWD
```

Specification data[36]

```
#define InSpecifierSet(kw,chain) (((1 << kw) & chain.KeywordSet) != 0)

extern SpecData InitSpecifiers      ELI_ARG((DefTableKey));
extern int NextSpecifier            ELI_ARG((TypeSpecifier, SpecData));
extern SpecData UpdateSpecification ELI_ARG((DefTableKey, TypeSpecifier, SpecData));
extern DefTableKey CompleteType        ELI_ARG((SpecData));

extern void BasicPointerTypes ELI_ARG((void));
extern void BasicPointerRefs  ELI_ARG((void));

#endif
```

This macro is attached to a product file.

Type.c[55]:

```
#include "pdl_gen.h"
#include "Typing.h"
#include "Expression.h"
#include "Type.h"

#define KWD(w,i,s) i
static int FSMInput[] = {
```
Specifier keywords[45]
```
};
#undef KWD

#define KWD(w,i,s) s
static long Exclude[] = {
```
Specifier keywords[45]
```
};
#undef KWD

static struct {int Next[7]; DefTableKey Type;} State[] = {
```
Finite-state machine[35]
```
};

SpecData
#ifdef PROTO_OK
InitSpecifiers(DefTableKey key)
#else
InitSpecifiers(key) DefTableKey key;
#endif
```
InitSpecifiers(DefTableKey key)[37]

```
int
#ifdef PROTO_OK
```

```
NextSpecifier(TypeSpecifier kw, SpecData chain)
#else
NextSpecifier(kw, chain) TypeSpecifier kw; SpecData chain;
#endif
```
NextSpecifier(TypeSpecifier kw, SpecData chain)[38]

```
SpecData
#ifdef PROTO_OK
UpdateSpecification(DefTableKey type, TypeSpecifier kw, SpecData chain)
#else
UpdateSpecification(type, kw, chain)
DefTableKey type; TypeSpecifier kw; SpecData chain;
#endif
```
UpdateSpecification(DefTableKey type, TypeSpecifier kw, SpecData chain)[39]

```
DefTableKey
#ifdef PROTO_OK
CompleteType(SpecData chain)
#else
CompleteType(chain) SpecData chain;
#endif
```
CompleteType(SpecData chain)[44]

Type key arrays[9]

```
void
#ifdef PROTO_OK
BasicPointerTypes(void)
#else
BasicPointerTypes()
#endif
```
void BasicPointerTypes(void)[7]

```
void
#ifdef PROTO_OK
BasicPointerRefs(void)
#else
BasicPointerRefs()
#endif
```
void BasicPointerRefs(void)[8]

This macro is attached to a product file.

Type.head[56]:

```
#include "Type.h"
```

This macro is attached to a product file.

Appendix A

Provide Readable Type Names

TypeName.specs[1]:

```
$/Tech/Strings.specs
```

This macro is attached to a product file.

TypeName.pdl[2]:

```
TypeName: CharPtr; "Strings.h"

TypeIs_void                  -> TypeName={"void"};
TypeIs_char                  -> TypeName={"char"};
TypeIs_signed_char           -> TypeName={"signed_char"};
TypeIs_unsigned_char         -> TypeName={"unsigned_char"};
TypeIs_short                 -> TypeName={"short"};
TypeIs_unsigned_short        -> TypeName={"unsigned_short"};
TypeIs_int                   -> TypeName={"int"};
TypeIs_NULL                  -> TypeName={"int 0"};
TypeIs_unsigned_int          -> TypeName={"unsigned_int"};
TypeIs_long                  -> TypeName={"long"};
TypeIs_unsigned_long         -> TypeName={"unsigned_long"};
TypeIs_float                 -> TypeName={"float"};
TypeIs_double                -> TypeName={"double"};
TypeIs_long_double           -> TypeName={"long_double"};
TypeIs_string                -> TypeName={"string"};

TypeIs_VoidPointer           -> TypeName={"void*"};
TypeIs_signed_charPointer    -> TypeName={"signed_char*"};
TypeIs_shortPointer          -> TypeName={"short*"};
TypeIs_intPointer            -> TypeName={"int*"};
TypeIs_longPointer           -> TypeName={"long*"};
TypeIs_unsigned_charPointer  -> TypeName={"unsigned_char*"};
TypeIs_unsigned_shortPointer -> TypeName={"unsigned_short*"};
TypeIs_unsigned_intPointer   -> TypeName={"unsigned_int*"};
TypeIs_unsigned_longPointer  -> TypeName={"unsigned_long *"};
TypeIs_floatPointer          -> TypeName={"float*"};
```

```
TypeIs_doublePointer          -> TypeName={"double*"};
TypeIs_long_doublePointer     -> TypeName={"long_double*"};
```

This macro is attached to a product file.

TypeName.lido[3]:

```
CLASS SYMBOL PointerDeclarator COMPUTE
  ResetTypeName(THIS.Type,"pointer");
END;

CLASS SYMBOL ArrayDeclarator COMPUTE
  ResetTypeName(THIS.Type,"array");
END;

CLASS SYMBOL FunctionDeclarator COMPUTE
  ResetTypeName(THIS.Type,"function");
END;

TREE SYMBOL struct_declaration_list COMPUTE
  ResetTypeName(THIS.Type,"struct");
END;

TREE SYMBOL enumerator_list COMPUTE
  ResetTypeName(THIS.Type,"enum");
END;
```

This macro is attached to a product file.