ANSI C Specification

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

January 19, 1998

Abstract

This document describes the parsing, name analysis and type analysis problems for ANSI C.

Contents

1 Scope

This specification describes the form and establishes certain properties of a program written in the C programming language. It formalizes the representation of C programs, the syntax of the C language, the correspondence between defining and applied occurrences of identifiers in a program, the structure of C data types, and the type of each expression in a program.

This specification can be processed by the Eli system to yield a "lint" program, a browsable HTML version of the document, or a PostScript version. The specification 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.

2 Normative references

The description here implements a portion of ANSI/ISO 9899-1990. It is organized in parallel with that document, for easy verification of the description.

This specification was developed and tested using Eli 4.0. A complete description of Eli, including the current public-domain source code, can be found at URL http://www.cs.colorado.edu/ eliuser/.

3 Definitions and Conventions

3.1 c.gla

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

c.gla[1] ≡

Lexical elements[15]
```
$#  (auxEOL)
$\f
```

This macro is attached to a product file.

3.2 c.con

A type-`con` file contains the context-free grammar describing the way a program is written.

c.con[2] ≡

Constants[43]
Enumeration constants[51]
String sequence[54]
Primary expressions[70]
Postfix operators[71]
Unary operators[79]
Cast operators[84]
Multiplicative operators[87]
Additive operators[89]
Bitwise shift operators[92]
Relational operators[94]
Equality operators[97]
Bitwise AND operator[100]
Bitwise exclusive OR operator[102]

Bitwise inclusive OR operator[104]
Logical AND operator[106]
Logical OR operator[109]
Conditional operator[112]
Assignment operators[115]
Comma operator[125]
Constant Expressions[126]
Declarations[127]
Statements[170]
External definitions[177]
Function definitions[178]
Optional symbols[14]

```
root:
  source .
source: / file .
file: translation_unit .
```

This macro is attached to a product file.

3.3 c.map

A type-`map` file describes the relationship between the phrase structure of the input text and the tree structure on which computations are carried out.

c.map[3] ≡

`MAPSYM`
Expressions[69]
Map the assignment operator symbol[116]
Declaration_specifier equivalence class[133]
Direct_declarator equivalence class[154]
Member_declarator equivalence class[139]
Parameter_part equivalence class[157]
Abstract_declarator equivalence class[166]

This macro is attached to a product file.

3.4 c.lido

A type-`lido` file describes computations to be carried out over the abstract syntax tree of a program.

c.lido[4] ≡

Function scope[18]
File scope[19]
Block scope[20]
Tag scope[24]
Function prototype scope[23]
Disambiguation of member names[28]
Semantics of declarations[129]
Semantics of structure and union specifiers[141]

Semantics of enumeration specifiers[143]
Semantics of declarators[159]
Access to the key attribute for ordinary identifiers[182]
Tags[144]
Use of an enumerated type tag[148]
Check for multiply-declared tags[145]
Tag that may or may not be a declaration[147]
Forward definition of a tag[149]
Anonymous structure, union or enumerated type[150]
Pointer declarators[161]
Array declarators[163]
Function declarators (including prototypes)[164]
Semantics of type names[167]
Semantics of function definitions[179]
Semantics of identifiers[180]

This macro is attached to a product file.

3.5 c.oil

A type-`oil` file describes the type model of a language.

c.oil[5] ≡

Types[30]
Implicit conversions[56]
Array subscripting[72]
Function calls[75]
Postfix increment and decrement operators[77]
Address and indirection operators[80]
Unary arithmetic operators[82]
Cast operator semantics[85]
Multiplicative operator semantics[88]
Additive operator semantics[90]
Bitwise shift operator semantics[93]
Relational operator semantics[95]
Equality operator semantics[98]
Bitwise AND operator semantics[101]
Bitwise exclusive OR operator semantics[103]
Bitwise inclusive OR operator semantics[105]
Logical AND operator semantics[107]
Logical OR operator semantics[110]
Conditional operator semantics[113]
Simple assignment[117]
Compound assignment[123]

This macro is attached to a product file.

3.6 c.c

Operational specifications of some characteristics are provided directly in a version of C that is compatible with C++. This code is provided with controls so that it can be compiled with non-ANSI C compilers.

c.c[6] ≡

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

State variable definitions[199]

Initial classification of identifier terminals[220]
Re-classify an identifier terminal to fit the context[221]
Bind a defining occurrence of an ordinary identifier[225]

```
void
InitOrdinary()
{
```
Initialize the array of definition table keys[209]
Initialize the set of bindings having file scope[185]
```
}
```

This macro is attached to a product file.

3.7 c.h

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

c.h[7] ≡

```
#ifndef C_H
#define C_H
#include "eliproto.h"
#include "envmod.h"
#include "RegionStack.h"
#include "OrdinaryIdStack.h"
#include "reparatur.h"
```

State variable declarations[204]

Operation interfaces[226]

```
extern void InitOrdinary();

#endif
```

This macro is attached to a product file.

Initialization of the scanner must include initialization of the environment for ordinary identifiers.

scanops.h[8] ≡

```
#ifndef SCANOPS_H
```

```
#define SCANOPS_H

#define SCANPTR \
  { InitOrdinary(); TokenEnd = TEXTSTART; StartLine = TokenEnd - 1; }

#endif
```

This macro is attached to a product file.

3.8 c.pdl

c.pdl[9] ≡

Properties of ordinary identifiers used during parsing[211]
Function conversion operator[63]
Property characterizing multiply-declared tags[146]

This macro is attached to a product file.

3.9 c.specs

c.specs[10] ≡

Name analysis modules for label identifiers[17]
Name analysis modules for member identifiers[140]
Name analysis module for ordinary identifiers[128]
Name spaces of identifiers[27]
Create a module to stack Environment values[184]
Create a module to stack DefTableKey values[206]
Create a module to stack identifier state values[214]

This macro is attached to a product file.

3.10 c.head

c.head[11] ≡

```
#include "c.h"
#include "termcode.h"
#include "IdStateStack.h"
```
Computations for obtaining the tag environment from a declarator[22]

This macro is attached to a product file.

3.11 c.init

c.init[12] ≡

Initialize the class of ordinary identifiers[215]

This macro is attached to a product file.

3.12 c.ctl

A type-`ctl` file contains directives controlling the behavior of Eli's attribute grammar analyzer.

c.ctl[13] $\equiv$

Do not allow attribution during tree construction[183]

This macro is attached to a product file.

4 Compliance

This specification describes only the C language. It makes no provision for either the library or the preprocessor.

5 Environment

The environmental considerations described here apply to the processor generated by Eli from this specification only. If the specification is used as a component of a larger specification, then some of these considerations may change.

5.1 Conceptual models

5.1.1 Translation environment

Program structure The processor described here deals with *translation units* – source files in which all preprocessor directives have been obeyed.

Translation phases The processor described here carries out only phases 5-7.

Diagnostics Error reports are issued for violations of the lexical and syntactic rules of the language, for invalid definitions and uses of identifiers, and for violations of the type rules.

5.1.2 Execution environments

Not covered by this specification.

5.2 Environmental considerations

5.2.1 Character sets

This specification assumes the properties of the characters common to the basic source and basic execution character sets that are stated in the standard.

Trigraph sequences Not covered by this specification.

Multibyte characters Not covered by this specification.

5.2.2 Character display semantics

The escapes described in this section are recognized by the processor, but their effect in output text is beyond the scope of the specification.

5.2.3 Signals and interrupts

Not covered by this specification.

5.2.4 Environmental limits

The translation environment described by this specification does not constrain the implementation.

6 Language

In the syntactic notation used here, syntactic categories (nonterminals) are indicated by words in `this` type, and literal words and character set members (terminals) by enclosing them 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. 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:

Optional symbols[14] ≡

```
abstract_declarator_opt: / abstract_declarator .
constant_exp_opt: / constant_expression .
declaration_list_opt: empty / declaration_list .
member_declarator_opt: / member_declarator .
expression_opt: / expression .
identifier_list_opt: / identifier_list .
init_declarator_list_opt: / init_declarator_list .
parameter_type_list_opt: /
  Prototype begin[186]  Begin a parameter_type_list[194] parameter_type_list
    End a parameter_type_list[195]  Prototype end[187]  .
statement_list_opt: / statement_list .

empty: .
```

This macro is invoked in definition 2.

The symbol `empty` is used only in those cases where the attribution rules require a tree node with a nonterminal child.

6.1 Lexical elements

Lexical elements[15] ≡

Identifiers[16]

Floating constants[44]
Integer constants[48]
Character constants[52]
String literals[53]
Comments[55]

This macro is invoked in definition 1.

6.1.1 Keywords

Keywords are described by literal terminals in the grammar.

6.1.2 Identifiers

Identifiers[16] ≡

```
identifier:  $[_a-zA-Z][_a-zA-Z0-9]* [IdnOrType]
```
Type definitions[168]
Additional terminal symbols representing identifiers[205]

This macro is invoked in definition 15.

The ambiguity resolution rules of Eli guarantee that every character sequence satisfying this definition will be initially classified as an `identifier`, and that the token processor `IdnOrType` will be invoked after the character sequence has been recognized. `IdnOrType` may then re-classify the sequence if that is appropriate.

Scopes of identifiers An identifier is *visible* (i.e., can be used) only within a region of the program text called its *scope*. There are four kinds of scopes: function, file, block and function prototype.

A label name is the only kind of identifier that has *function scope*. It can be used (in a `goto` statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance followed by a `:`. Label names shall be unique within a function.

An instantiation of Eli's `Unique` module is used to verify that label names must be unique within a function.

Name analysis modules for label identifiers[17] ≡

```
$/Prop/Unique.gnrc +instance=Label +referto=Label :inst
```

This macro is invoked in definition 10.

The `Unique` module exports two symbols: `LabelUnique` marks the nodes representing the occurrences of the identifiers to be tested for uniqueness, while `LabelRangeUnique` marks the subtree containing all of those nodes.

Function scope[18] ≡

```
SYMBOL file                INHERITS LabelRootScope, LabelRangeUnique END;
SYMBOL function_definition INHERITS LabelRangeScope                  END;

RULE: jump_statement ::= 'goto' LabelUse ';' END;

SYMBOL LabelUse INHERITS LabelIdUseEnv COMPUTE
  IF(EQ(THIS.LabelKey,NoKey),
    message(ERROR, "Label identifier is not defined", 0, COORDREF));
```

```
END;

RULE: labeled_statement ::= LabelDef ':' statement END;

SYMBOL LabelDef INHERITS LabelIdDefScope, LabelUnique COMPUTE
  IF(NOT(THIS.LabelUnique),
    message(ERROR, "Label identifier is multiply defined", 0, COORDREF));
END;
```

This macro is invoked in definition 4.

Every other identifier has a scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit.

File scope[19] ≡

```
SYMBOL file INHERITS TagRootScope END;
```

This macro is invoked in definition 4.

If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the `}` that closes the associated block.

Block scope[20] ≡

```
SYMBOL declaration_list_opt INHERITS TagRangeScope END;
SYMBOL compound_statement   INHERITS TagRangeScope END;
ATTR TagEnv: Environment;

RULE: function_definition ::=
  declaration_specifiers declarator declaration_list_opt compound_statement
COMPUTE
  Block scope tag environment for a function definition[21]
END;

RULE: function_definition ::=
                         declarator declaration_list_opt compound_statement
COMPUTE
  Block scope tag environment for a function definition[21]
END;
```

This macro is invoked in definition 4.

In a `function_definition`, the `declaration_list_opt` and `compound_statement` are parts of the same scope. If a prototype is given for the function, that prototype is also part of the scope.

Block scope tag environment for a function definition[21] ≡

```
.TagEnv=
  declarator CONSTITUENTS parameter_part.TagEnv
    SHIELD (parameter_part, constant_exp_opt)
    WITH (Environment, Leftmost, IDENTICAL, NOENV);
declaration_list_opt.TagEnv=IF(EQ(.TagEnv, NoEnv), NewEnv(), .TagEnv);
compound_statement.TagEnv=declaration_list_opt.TagEnv;
```

This macro is invoked in definition 20.

A `parameter_part` phrase may contain nested `parameter_part` phrases, which could not be the prototype of the function being called. Thus the `CONSTITUENTS` process must be prevented from examining components of `parameter_part` phrases. On the other hand, it must examine nested `declarator` phrases because the function and its prototype may be enclosed in parentheses. The `SHIELD` clause has this effect, and also prevents the examination of array size specifications (which cannot contain the function's prototype).

Even at the top level, however, a `declarator` may contain many `parameter_part` phrases. The one defining the parameters of the function is textually the leftmost, so the `WITH` clause uses `Leftmost` to choose the leftmost non-null environment.

Computations for obtaining the tag environment from a declarator[22] ≡

```
#define Leftmost(x,y) ((x)==NoEnv?(y):(x))
#define NOENV() NoEnv
```

This macro is invoked in definition 11.

If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator.

Function prototype scope[23] ≡

```
SYMBOL parameter_part INHERITS TagRangeScope END;
```

This macro is invoked in definition 4.

If an outer declaration of a lexically identical identifier exists in the same name space, it is hidden until the current scope terminates, after which it again becomes visible.

Two identifiers have the same scope if and only if their scopes terminate at the same point.

Structure, union and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag.

Tag scope[24] ≡

```
SYMBOL TagDef INHERITS TagIdDefScope END;
SYMBOL TagUse INHERITS TagIdUseEnv   END;
```

This macro is invoked in definition 4.

Each enumeration constant has scope that begins just after the appearance of the defining `enumerator` in an `enumerator_list`.

enumerator[25] ≡

`enumerator` *Carry out a deferred binding*[223]

This macro is invoked in definition 142.

Any other identifier has scope that begins just after the completion of its declarator.

Innermost declarator[26] ≡

`direct_declarator` *Carry out a deferred binding*[223]

This macro is invoked in definition 153.

A `typedef_name` is an ordinary identifier, declared by a `declaration` containing the `storage_class_specifier` "typedef". The scopes of all ordinary identifiers must therefore be determined during parsing in order to support `IdnOrType` in its task of classifying character sequences recognized as `identifier`. Only in this way is it possible to decide whether a particular use of an ordinary identifier should be interpreted as a `typedef_name`.

Linkages of identifiers

Name spaces of identifiers If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

Name spaces of identifiers[27] ≡

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

Ordinary identifier name space[181]

This macro is invoked in definition 10.

Disambiguation of member names[28] ≡

```
RULE: Expression ::= Expression '.' MemberIdUse END;
RULE: Expression ::= Expression '->' MemberIdUse END;
```

This macro is invoked in definition 4.

Ordinary identifiers must be renamed during parsing in order to avoid an ambiguity in the grammar. That process does not meet the preconditions for using any of the Eli name analysis modules, and therefore must be handled with more primitive operations. The details are presented in a later chapter.

component_list: [29] ≡

```
component_list:
```

Begin a component_list[192] `'{' struct_declaration_list`

End a component_list[193] `'}' .`

This macro is invoked in definition 138.

Storage duration of objects

Types 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 introduces the OIL identifiers used to represent 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.

The standard provides mechanisms for a user to define additional types. Each of these mechanisms is specified in this section by an OIL `CLASS`.

Types[30] ≡

> *Signed integer types*[31]
> *Unsigned integer types*[32]
> *Floating types*[33]
> *Integral types*[40]
> *Arithmetic types*[41]
> *Scalar types*[42]
> *Promoted types*[59]
> *Structure types*[36]
> *Union types*[37]
> *Enumeration types*[34]
> *Function type derivation*[38]
> *Array type derivation*[35]
> *Pointer type derivation*[39]

This macro is invoked in definition 5.

A `TypeIs_char` object is large enough to store any member of the basic execution character set.

There are four *signed integer types*:

Signed integer types[31] ≡

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

This macro is invoked in definition 30.

For each of the signed integer types, there is a corresponding (but different) *unsigned integer type* that uses the same amount of storage (including sign information) and has the same alignment requirements:

Unsigned integer types[32] ≡

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

This macro is invoked in definition 30.

There are three *floating types*:

Floating types[33] ≡

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

This macro is invoked in definition 30.

`TypeIs_char`, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.

An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.

Enumeration types[34] ≡

```
CLASS TypeIs_Enum() BEGIN
  Enumeration conversion[58]
  Operator defined for this enumeration type[122]
END;
```

This macro is invoked in definition 30.

`TypeIs_void` comprises an empty set of values; it is an incomplete type that cannot be completed.

Any number of *derived types* can be constructed from the object, function and incomplete types, as follows:

An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. Array objects are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is `T`, the array type is sometimes called "array of `T`". The construction of an array type from an element type is called "array type derivation".

Array type derivation[35] ≡

```
CLASS TypeIs_Array(elementType, pointerType) BEGIN
  Array conversion[62]
  Operators defined for this array type[73]
END;
```

This macro is invoked in definition 30.

A *structure type* describes a sequentially allocated nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

Structure types[36] ≡

```
CLASS TypeIs_Struct() BEGIN
  Operator defined for this struct type[120]
END;
```

This macro is invoked in definition 30.

A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

Union types[37] ≡

```
CLASS TypeIs_Union() BEGIN
  Operator defined for this union type[121]
END;
```

This macro is invoked in definition 30.

A *function type* describes a function with a specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is `T`, the function type is sometimes called "function returning `T`". The construction of a function type from a return type is called "function type derivation".

Function type derivation[38] ≡

```
CLASS TypeIs_Function(returnType) BEGIN
  Operators defined for this function type[76]
END;
```

This macro is invoked in definition 30.

A *pointer type* may be derived from a function type, an object type, or an incomplete type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type `T` is sometimes called "pointer to `T`". The construction of a pointer type from a referenced type is called "pointer type derivation".

Pointer type derivation[39] ≡

```
CLASS TypeIs_Pointer(referencedType) BEGIN
  Pointer[66]
  Discard pointer values[65]
  Null constant for this pointer type[68]
  Operators defined for this pointer type[74]
END;
```

This macro is invoked in definition 30.

These methods of constructing types can be applied recursively.

The standard specifies that the representations of *integral* objects shall define values by use of a pure binary numeration system:

Integral types[40] ≡

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

This macro is invoked in definition 30.

Enumerated types are considered to be integral also, but an OIL set is a static object and therefore cannot contain a class as a member. Each enumerated type must therefore be individually described as integral, which is the effect of the integer promotions described below.

Arithmetic objects are those that can participate in arithmetic operations:

Arithmetic types[41] ≡

```
SET TypeIs_Arithmetic = TypeIs_Integral + TypeIs_Floating;
```

This macro is invoked in definition 30.

Scalar objects are those that can be compared with one another:

Scalar types[42] ≡

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

This macro is invoked in definition 30.

Pointer types are considered to be scalar also, but an OIL set is a static object and therefore cannot contain a class as a member. Each pointer type must therefore be individually described as scalar, which is the effect of the pointer conversions described below.

Compatible type and composite type

6.1.3 Constants

There is no `enumeration_constant` because an `enumeration_constant` is recognized as an `identifier`. If `enumeration_constant` appeared in this definition of `constant`, there would be an LALR(1) conflict in the grammar.

Constants[43] ≡

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

This macro is invoked in definition 2.

Floating constants *Floating constants*[44] ≡

floating_constant: $(*F*[45]*E*[46]?|*D*[47]+*E*[46])[flFL]? [mkidn]

This macro is invoked in definition 15.

F[45] ≡
(*D*[47]*\.*D*[47]+) This macro is invoked in definition 44.

E[46] ≡
([eE][+-]?*D*[47]+) This macro is invoked in definition 44.

D[47] ≡
[0-9] This macro is invoked in definitions 44, 45, 46, and 48.

Integer constants *Integer constants*[48] ≡

integer_constant: $([1-9]*D*[47]*|*O*[49]|*H*[50])([uU][lL]?|[lL][uU]?)? [c_mkint]

This macro is invoked in definition 15.

O[49] ≡
0[0-7]* This macro is invoked in definition 48.

H[50] ≡
0[xX][0-9a-fA-F]+ This macro is invoked in definition 48.

Enumeration constants Enumeration constants are identifiers, so no additional lexical specification is required.

Enumeration constants[51] ≡

enumeration_constant: OrdinaryIdDef *Defer binding the declared identifier*[222] .

This macro is invoked in definition 2.

Character constants C character denotations are defined by a canned description in Eli.

Character constants[52] ≡

```
character_constant:  C_CHAR_CONSTANT
```

This macro is invoked in definition 15.

6.1.4 String literals

C string literals are defined by a canned description in Eli.

String literals[53] ≡

```
string_literal:  C_STRING_LIT
```

This macro is invoked in definition 15.

Sequences of string literals are translated as single strings. By defining such a sequence as a `StringSeq`, we allow the specification to use Eli's canned description of a single string literal.

String sequence[54] ≡

```
StringSeq:
  string_literal /
  StringSeq string_literal .
```

This macro is invoked in definition 2.

6.1.5 Operators

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

6.1.6 Punctuators

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

6.1.7 Header names

Preprocessing is not done within the generated compiler.

6.1.8 Preprocessing numbers

Preprocessing is not done within the generated compiler.

6.1.9 Comments

C comments are defined by a canned description in Eli.

Comments[55] ≡

```
  C_COMMENT
```

This macro is invoked in definition 15.

6.2 Conversions

Several operators convert operands from one type to another automatically. This subclause specifies the result required from such *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*).

Each implicit conversion is defined in this section by an OIL `COERCION` specification.

Implicit conversions[56] ≡

> *Characters and integers*[57]
> *Usual arithmetic conversions*[60]
> *void*[64]
> *Conversions for null pointer constants*[67]

This macro is invoked in definition 5.

The specifications in this section assume that objects of certain types are capable of representing all of the values of other types. Those assumptions are completely compatible with the standard, but may not hold for a specific implementation of C. If this type analysis module is being used in a translator, and the assumptions do not hold, then the specifications must be changed as indicated in the accompanying text.

6.2.1 Arithmetic operands

Characters and integers A `char`, a `short`, or an `int` bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an `int` or `unsigned int` may be used. If a `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the *integral promotions*. All other arithmetic types are unchanged by the integral promotions.

Characters and integers[57] ≡

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

This macro is invoked in definition 56.

Enumeration conversion[58] ≡

```
COERCION CEnumtoInt(TypeIs_Enum): TypeIs_int;
```

This macro is invoked in definition 34.

It may be necessary to alter these definitions based upon the target machine if this module is to be used in a translator: If an `int` object cannot represent all values of the argument type, the result type for those conversions must be `unsigned int`.

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:

Promoted types[59] ≡

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

This macro is invoked in definition 30.

Signed and unsigned integers

Floating and integral

Floating types

Usual arithmetic conversions Many binary operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

Usual arithmetic conversions[60] ≡

```
COERCION CDoubletoLongDouble(TypeIs_double): TypeIs_long_double;
COERCION CFloattoDouble(TypeIs_float): TypeIs_double;
COERCION CUnsignedLongtoFloat(TypeIs_unsigned_long): TypeIs_float;
Integer conversions[61]
```

This macro is invoked in definition 56.

If both operands are integral, then integral promotions are performed on them. Then the following rules are applied:

Integer conversions[61] ≡

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

This macro is invoked in definition 60.

It may be necessary to alter the definition of `CUnsignedInttoLong` based upon the target machine if this module is to be used in a translator: If a `TypeIs_long` object cannot represent all values of type `TypeIs_unsigned_int` the result type must be `TypeIs_unsigned_long`.

6.2.2 Other operands

Lvalues and function designators Except when it is the operand of the `sizeof` operator or the unary `&` operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with `wchar_t`, an lvalue that has type "array of *type*" is converted to an expression that has type "pointer to *type*" that points to the initial element of the array and is not an lvalue.

Array conversion[62] ≡

```
COERCION CArraytoPtr (TypeIs_Array): pointerType;
```

This macro is invoked in definition 35.

Except when it is the operand of the `sizeof` operator or the unary `&` operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*".

Function conversion operator[63] ≡

```
CFunctoPtr;
```

This macro is invoked in definition 9.

void The (nonexistent) value of a *void expression* (an expression that has `TypeIs_void`) shall not be used in any way, and implicit or explicit conversions (except to `TypeIs_void`) shall not be applied to such an expression. If an expression of any other type occurs in a context where a void expression is required, its value or designator is discarded. (A void expression is evaluated for its side effects.)

void[64] ≡

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

This macro is invoked in definition 56.

Discard pointer values[65] ≡

```
COERCION CPtrtoVoid(TypeIs_Pointer): TypeIs_void;
```

This macro is invoked in definition 39.

Pointer A `TypeIs_VoidPointer` may be converted to or from a pointer to any incomplete or object type. In this specification, the conversion to a `TypeIs_VoidPointer` may be either implicit or explicit; the conversion from `TypeIs_VoidPointer` must be explicit.

Pointer[66] ≡

```
COERCION CVoidPtr(TypeIs_Pointer): TypeIs_VoidPointer;
```

This macro is invoked in definition 39.

An integral constant expression with the value 0, or such an expression cast to `TypeIs_VoidPointer`, is called a *null pointer constant*. If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function. A null pointer constant has `TypeIs_NULL`.

Conversions for null pointer constants[67] ≡

```
COERCION CNulltoVoidPtr(TypeIs_NULL): TypeIs_VoidPointer;
COERCION CNulltoIntegral(TypeIs_NULL): TypeIs_Integral;
```

This macro is invoked in definition 56.

(The implicit conversion `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.)

Null constant for this pointer type[68] ≡

```
COERCION CNulltoPtr(TypeIs_NULL): TypeIs_Pointer;
```

This macro is invoked in definition 39.

6.3 Expressions

A C expression is a construct that yields a value. The standard distinguishes expressions on the basis of the precedence of that expression's operator. This is done both to simplify the explanation of the meaning of an expression and to preserve the context in which an expression occurs so that the language's precedence and association rules are obeyed. By describing the expressions in this way, the standard avoids the need to discuss the concepts of precedence and association directly.

Precedence and association are only interesting because they relate the structure of the program tree to the linear representation of the text. Once the tree has been built, the operands of an operator are reflected in the tree structure and there is no longer any need to distinguish different kinds of expression. Thus we define an equivalence class consisting of all of the symbols used by the standard to name expressions:

Expressions[69] ≡

```
Expression ::=
  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
  constant_expression expression expression_opt .
```

This macro is invoked in definition 3.

Any symbol in this equivalence class will be represented in the tree by a node with the name `Expression`, which is a new symbol not appearing anywhere in the grammar.

6.3.1 Primary expressions

`IdUse` replaces the symbol `identifier` appearing in the standard's definition of a `primary_expression`. As discussed in Section 1.1.2 above, `IdUse` makes the syntactic context of an ordinary identifier use explicit.

`StringSeq` replaces the symbol `string_literal` appearing in the standard's definition of a `primary_expression`. As discussed in Section 1.1.4 above, `StringSeq` embodies the semantic condition that a sequence of string literals is considered to be a single string literal in C.

Primary expressions[70] ≡

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

IdUse:
  UnboundIdUse /
  OrdinaryIdUse .
```

This macro is invoked in definition 2.

6.3.2 Postfix operators

Postfix operators[71] ≡

```
postfix_expression:
  primary_expression /
  postfix_expression '[' expression ']' /
  postfix_expression '(' argument_exp_list ')' /
  postfix_expression '('  ')' /
  postfix_expression '.' identifier /
```

```
  postfix_expression '->' identifier /
  postfix_expression '++' /
  postfix_expression '--' .

argument_exp_list:
  assignment_expression /
  argument_exp_list ',' assignment_expression.
```

This macro is invoked in definition 2.

Array subscripting *Array subscripting*[72] ≡

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

This macro is invoked in definition 5.

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 defined for this array type[73] ≡

```
OPER Array_Subscript_Op(TypeIs_Array, TypeIs_unsigned_long): elementType;
```

This macro is defined in definitions 73 and 118.
This macro is invoked in definition 35.

Operators defined for this pointer type[74] ≡

```
OPER Subscript_Op(TypeIs_Pointer, TypeIs_unsigned_long): referencedType;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

Function calls *Function calls*[75] ≡

```
INDICATION Call_Indication: FunCall_Op;
```

This macro is invoked in definition 5.

A distinct call operator is defined for each function type.

Operators defined for this function type[76] ≡

```
OPER FunCall_Op(TypeIs_Function): returnType;
```

This macro is invoked in definition 38.

Postfix increment and decrement operators *Postfix increment and decrement operators*[77] ≡

```
INDICATION Increment_Indication: Increment_Op, Ptr_Inc_Op;
INDICATION Decrement_Indication: Decrement_Op, Ptr_Dec_Op;
OPER Increment_Op, Decrement_Op(TypeIs_Arithmetic): TypeIs_Arithmetic;
```

This macro is invoked in definition 5.

Distinct increment and decrement operators are defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[78] ≡

```
OPER Ptr_Inc_Op, Ptr_Dec_Op(TypeIs_Pointer): TypeIs_Pointer;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

6.3.3 Unary operators

Unary operators[79] ≡

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

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

This macro is invoked in definition 2.

Prefix increment and decrement operators No distinctions are made in this specification between the prefix and postfix operations. The only difference in semantics lies in their relationship to the reference.

Address and indirection operators *Address and indirection operators*[80] ≡

```
INDICATION Dereference_Indication: Ptr_Deref_Op;
INDICATION Reference_Indication: Ptr_Ref_Op;
```

This macro is invoked in definition 5.

A distinct indirection operator is defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[81] ≡

```
OPER Ptr_Deref_Op(TypeIs_Pointer): referencedType;
OPER Ptr_Ref_Op(referencedType): TypeIs_Pointer;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

Unary arithmetic operators *Unary arithmetic operators*[82] ≡

```
INDICATION Plus_Indication: Plus_Op;
INDICATION Minus_Indication: Minus_Op;
INDICATION Bitwise_Not_Indication: Bitwise_Not_Op;
INDICATION Not_Indication: Not_Op, Void_Ptr_Not_Op, Ptr_Not_Op;
OPER Plus_Op, Minus_Op(TypeIs_ArithPromoted): TypeIs_ArithPromoted;
OPER Bitwise_Not_Op(TypeIs_IntegralPromoted): TypeIs_IntegralPromoted;
OPER Not_Op(TypeIs_Scalar): TypeIs_int;
OPER Void_Ptr_Not_Op (TypeIs_VoidPointer): TypeIs_int;
```

This macro is invoked in definition 5.

A distinct logical negation operator is defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[83] ≡

```
OPER Ptr_Not_Op (TypeIs_Pointer): TypeIs_int;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

6.3.4 Cast operators

Cast operators[84] ≡

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

This macro is invoked in definition 2.

Cast operator semantics[85] ≡

```
INDICATION Cast_Indication: Cast_Op, Cast_IntegraltoPtr, Cast_VoidPtrtoPtr;

SET TypeIs_CastResult = TypeIs_Scalar;
OPER Cast_Op(TypeIs_Scalar): TypeIs_CastResult;
```

This macro is invoked in definition 5.

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.

Operators defined for this pointer type[86] ≡

```
OPER Cast_IntegraltoPtr(TypeIs_unsigned_long): TypeIs_Pointer;
OPER Cast_VoidPtrtoPtr(TypeIs_VoidPointer): TypeIs_Pointer;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

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.)

6.3.5 Multiplicative operators

Multiplicative operators[87] ≡

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

This macro is invoked in definition 2.

Multiplicative operator semantics[88] ≡

```
INDICATION Multiplication_Indication: MulOp;
INDICATION Division_Indication: DivOp;
INDICATION Mod_Indication: ModOp;
OPER MulOp, DivOp(TypeIs_Arithmetic, TypeIs_Arithmetic): TypeIs_Arithmetic;
OPER ModOp(TypeIs_Integral, TypeIs_Integral): TypeIs_Integral;
```

This macro is invoked in definition 5.

6.3.6 Additive operators

Additive operators[89] ≡

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

This macro is invoked in definition 2.

Additive operator semantics[90] ≡

```
INDICATION Addition_Indication:
  AddOp, Void_Ptr_Add_Op, Void_Ptr_Rev_Add_Op, Ptr_Add_Op, Ptr_Rev_Add_Op;
INDICATION Subtraction_Indication:
  SubOp, Void_Ptr_Sub_Op, Ptr_Sub_Op, Ptr_Ptr_Sub_Op;
OPER AddOp, SubOp(TypeIs_Arithmetic, TypeIs_Arithmetic): TypeIs_Arithmetic;
OPER
  Void_Ptr_Add_Op, Void_Ptr_Sub_Op(TypeIs_VoidPointer, TypeIs_unsigned_long):
    TypeIs_VoidPointer;
```

```
OPER
  Void_Ptr_Rev_Add_Op(TypeIs_unsigned_long, TypeIs_VoidPointer):
    TypeIs_VoidPointer;
```

This macro is invoked in definition 5.

Distinct additive operators are defined for each pointer type other than TypeIs_VoidPointer.

Operators defined for this pointer type[91] ≡

```
OPER
  Ptr_Add_Op, Ptr_Sub_Op(TypeIs_Pointer, TypeIs_unsigned_long): TypeIs_Pointer;
OPER
  Ptr_Rev_Add_Op, Ptr_Rev_Sub_Op(TypeIs_unsigned_long, TypeIs_Pointer):
    TypeIs_Pointer;
OPER Ptr_Ptr_Sub_Op(TypeIs_Pointer, TypeIs_Pointer): TypeIs_unsigned_long;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

The integral operand of an additive operator involving a pointer and an integer is specified to be a TypeIs_unsigned_long. 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.

6.3.7 Bitwise shift operators

Bitwise shift operators[92] ≡

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

This macro is invoked in definition 2.

Bitwise shift operator semantics[93] ≡

```
INDICATION Bit_Shift_Left_Indication: Bit_Shift_Left_Op;
INDICATION Bit_Shift_Right_Indication: Bit_Shift_Right_Op;

SET TypeIs_ShiftCount = TypeIs_IntegralPromoted;
OPER Bit_Shift_Right_Op, Bit_Shift_Left_Op
  (TypeIs_IntegralPromoted, TypeIs_ShiftCount): TypeIs_IntegralPromoted;
```

This macro is invoked in definition 5.

By defining TypeIs_ShiftCount as being equal to TypeIs_IntegralPromoted and then using these two set identifiers in defining the bitwise shift operators, the specification allows different types of operands for the shift count and the value being shifted. The type of the result is that of the promoted left operand.

6.3.8 Relational operators

Relational operators[94] ≡

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

This macro is invoked in definition 2.

Relational operator semantics[95] ≡

```
INDICATION LessThan_Indication: LessThan_Op, Ptr_LT_Op;
INDICATION Greater_Indication: Greater_Op, Ptr_GT_Op;
INDICATION LessThan_Equal_Indication: LessThan_Equal_Op, Ptr_LTE_Op;
INDICATION Greater_Equal_Indication: Greater_Equal_Op, Ptr_GTE_Op;
OPER Greater_Op, LessThan_Op, Greater_Equal_Op, LessThan_Equal_Op
  (TypeIs_Scalar, TypeIs_Scalar): TypeIs_int;
```

This macro is invoked in definition 5.

Distinct relational operators are defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[96] ≡

```
OPER Ptr_LT_Op, Ptr_GT_Op, Ptr_LTE_Op, Ptr_GTE_Op
  (TypeIs_Pointer, TypeIs_Pointer): TypeIs_int;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

6.3.9 Equality operators

Equality operators[97] ≡

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

This macro is invoked in definition 2.

Equality operator semantics[98] ≡

```
INDICATION Equality_Indication: Equality_Op, Ptr_Eq_Op;
INDICATION Not_Equal_Indication: Not_Equal_Op, Ptr_NEq_Op;
OPER Equality_Op, Not_Equal_Op(TypeIs_Scalar, TypeIs_Scalar): TypeIs_int;
```

This macro is invoked in definition 5.

Distinct equality operators are defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[99] ≡

```
OPER Ptr_Eq_Op, Ptr_NEq_Op(TypeIs_Pointer, TypeIs_Pointer): TypeIs_int;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

6.3.10 Bitwise AND operator

Bitwise AND operator[100] ≡

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

This macro is invoked in definition 2.

Bitwise AND operator semantics[101] ≡

```
INDICATION Bitwise_And_Indication: Bitwise_And_Op;
OPER Bitwise_And_Op(TypeIs_Integral, TypeIs_Integral): TypeIs_Integral;
```

This macro is invoked in definition 5.

6.3.11 Bitwise exclusive OR operator

Bitwise exclusive OR operator[102] ≡

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

This macro is invoked in definition 2.

Bitwise exclusive OR operator semantics[103] ≡

```
INDICATION Bitwise_XOr_Indication: Bitwise_XOr_Op;
OPER Bitwise_XOr_Op(TypeIs_Integral, TypeIs_Integral): TypeIs_Integral;
```

This macro is invoked in definition 5.

6.3.12 Bitwise inclusive OR operator

Bitwise inclusive OR operator[104] ≡

```
inclusive_OR_expression:
  exclusive_OR_expression /
  inclusive_OR_expression '|' exclusive_OR_expression.
```

This macro is invoked in definition 2.

Bitwise inclusive OR operator semantics[105] ≡

```
INDICATION Bitwise_Or_Indication: Bitwise_Or_Op;
OPER Bitwise_Or_Op(TypeIs_Integral, TypeIs_Integral): TypeIs_Integral;
```

This macro is invoked in definition 5.

6.3.13 Logical AND operator

Logical AND operator[106] ≡

```
logical_AND_expression:
  inclusive_OR_expression /
  logical_AND_expression '&&' inclusive_OR_expression.
```

This macro is invoked in definition 2.

Logical AND operator semantics[107] ≡

```
INDICATION And_Indication: And_Op, Ptr_And_Op;
OPER And_Op(TypeIs_Scalar, TypeIs_Scalar): TypeIs_int;
```

This macro is invoked in definition 5.

Distinct logical AND operators are defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[108] ≡

```
OPER Ptr_And_Op(TypeIs_Pointer, TypeIs_Pointer): TypeIs_int;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

6.3.14 Logical OR operator

Logical OR operator[109] ≡

```
logical_OR_expression:
  logical_AND_expression /
  logical_OR_expression '||' logical_AND_expression.
```

This macro is invoked in definition 2.

Logical OR operator semantics[110] ≡

```
INDICATION Or_Indication: Or_Op, Ptr_Or_Op;
OPER Or_Op(TypeIs_Scalar, TypeIs_Scalar): TypeIs_int;
```

This macro is invoked in definition 5.

Distinct logical OR operators are defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[111] ≡

```
OPER Ptr_Or_Op(TypeIs_Pointer, TypeIs_Pointer): TypeIs_int;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

6.3.15 Conditional operator

Conditional operator[112] ≡

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

This macro is invoked in definition 2.

Conditional operator semantics[113] ≡

```
INDICATION Conditional_Indication: Conditional_Op, Ptr_Conditional_Op;
OPER Conditional_Op(TypeIs_Scalar): TypeIs_int;
```

This macro is invoked in definition 5.

Distinct conditional operators are defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[114] ≡

```
OPER Ptr_Conditional_Op(TypeIs_Pointer): TypeIs_int;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

This specification constrains only the first operand of the conditional. Temporarily, the specification will simply balance the second and third operands of the conditional to obtain the result type.

6.3.16 Assignment operators

Assignment operators[115] ≡

```
assignment_expression:
  conditional_expression /
  unary_expression assignment_operator assignment_expression .

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

This macro is invoked in definition 2.

An assignment operator is identified during semantic analysis in the same way as any other dyadic operator. It is therefore convenient to map the symbol `assignment_operator` to `binary_operator` in the tree:

Map the assignment operator symbol[116] ≡

```
binary_operator ::= assignment_operator .
```

This macro is invoked in definition 3.

Simple assignment *Simple assignment*[117] ≡

```
INDICATION Assign_Indication: Assign_Op, Ptr_Assign_Op, Enum_Assign_Op,
Struct_Assign_Op, Union_Assign_Op, Ptr_Void_Assign_Op, Array_Assign_Op;

SET TypeIs_RHS_Arithmetic = TypeIs_Arithmetic;
OPER Assign_Op(TypeIs_Arithmetic, TypeIs_RHS_Arithmetic): TypeIs_Arithmetic;
```

This macro is invoked in definition 5.

By defining `TypeIs_RHS_Scalar` as being equal to `TypeIs_Scalar` and then using these two set identifiers in defining `Assign_Op`, the specification allows different types of operands for the left and right operands. The type of the result is that of the left operand.

Distinct simple assignments are defined for each array type.

Operators defined for this array type[118] ≡

```
OPER Array_Assign_Op(TypeIs_Array, pointerType): TypeIs_Array;
```

This macro is defined in definitions 73 and 118.
This macro is invoked in definition 35.

Distinct simple assignments are defined for each pointer type other than `TypeIs_VoidPointer`.

Operators defined for this pointer type[119] ≡

```
OPER Ptr_Assign_Op(TypeIs_Pointer, TypeIs_Pointer): TypeIs_Pointer;
OPER Ptr_Void_Assign_Op(TypeIs_Pointer, TypeIs_VoidPointer): TypeIs_Pointer;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

Operator defined for this struct type[120] ≡

```
OPER Struct_Assign_Op(TypeIs_Struct, TypeIs_Struct): TypeIs_Struct;
```

This macro is invoked in definition 36.

Operator defined for this union type[121] ≡

```
OPER Union_Assign_Op(TypeIs_Union, TypeIs_Union): TypeIs_Union;
```

This macro is invoked in definition 37.

Operator defined for this enumeration type[122] ≡

```
OPER Enum_Assign_Op(TypeIs_Enum, TypeIs_int): TypeIs_Enum;
```

This macro is invoked in definition 34.

Compound assignment *Compound assignment*[123] ≡

```
INDICATION Mult_Eq_Indication: Mult_Eq_Op;
INDICATION Div_Eq_Indication: Div_Eq_Op;
INDICATION Mod_Eq_Indication: Mod_Eq_Op;
INDICATION Plus_Eq_Indication: Plus_Eq_Op, Ptr_Plus_Eq_Op;
```

```
INDICATION Minus_Eq_Indication: Minus_Eq_Op, Ptr_Minus_Eq_Op;
INDICATION Bitwise_Shift_Left_Eq_Indication: Bitwise_Shift_Left_Eq_Op;
INDICATION Bitwise_Shift_Right_Eq_Indication: Bitwise_Shift_Right_Eq_Op;
INDICATION Bitwise_And_Eq_Indication: Bitwise_And_Eq_Op;
INDICATION Bitwise_XOr_Eq_Indication: Bitwise_XOr_Eq_Op;
INDICATION Bitwise_Or_Eq_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;
OPER 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 invoked in definition 5.

Distinct compound assignments are defined for each pointer type other than `TypeIs_VoidPointer`. The integral operand of a compound assignment involving a pointer and an integer is specified to be a `TypeIs_unsigned_long`. 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 defined for this pointer type[124] ≡

```
OPER Ptr_Plus_Eq_Op(TypeIs_Pointer, TypeIs_unsigned_long): TypeIs_Pointer;
OPER Ptr_Minus_Eq_Op(TypeIs_Pointer, TypeIs_unsigned_long): TypeIs_Pointer;
```

This macro is defined in definitions 74, 78, 81, 83, 86, 91, 96, 99, 108, 111, 114, 119, and 124.
This macro is invoked in definition 39.

6.3.17 Comma operator

Comma operator[125] ≡

```
expression:
  assignment_expression /
  expression ',' assignment_expression .
```

This macro is invoked in definition 2.

6.4 Constant Expressions

Constant Expressions[126] ≡

```
constant_expression:
  conditional_expression .
```

This macro is invoked in definition 2.

6.5 Declarations

The parsing techniques used to resolve a syntactic ambiguity involving `typedef` require a more complex definition for `declaration_specifiers` than that given in the standard. We give that definition in conjunction with our presentation of the parsing method.

Declarations[127] ≡

```
declaration:
  declaration_specifiers init_declarator_list_opt ';'
  End of a declaration[216] .

declaration_specifiers: [136]

init_declarator_list:
  init_declarator /
  init_declarator_list ',' init_declarator .

init_declarator:
  declarator /
  declarator '=' initializer .

Storage-class specifiers[132]
Type specifiers[134]
Type qualifiers[151]
Declarators[153]
Type names[165]
Initialization[169]
```

This macro is invoked in definition 2.

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 the same name space, except for tags as specified in 6.5.2.3. An instance of the Eli `Unique` module can be used to detect multiply defined identifiers.

Name analysis module for ordinary identifiers[128] ≡

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

This macro is invoked in definition 10.

Semantics of declarations[129] ≡

```
SYMBOL file INHERITS RangeUnique END;

RULE: declaration_specifiers   LISTOF declaration_specifier END;
RULE: init_declarator_list     LISTOF init_declarator       END;
```

This macro is defined in definitions 129, 130, and 131.
This macro is invoked in definition 4.

A `declaration` specifies the interpretation and attributes of a set of identifiers.

`declaration_specifiers` consists of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The type that they name is embodied in the `TypeSpecified` attribute of the `declaration_specifiers` symbol. Information gleaned from the declaration specifiers about the storage class and type qualifier is embodied in six integer attributes that have the value `1` if the corresponding specifier or qualifier was present and `0` otherwise:

Semantics of declarations[130] ≡

```
SYMBOL declaration_specifiers: TypeSpecified: DefTableKey;
ATTR   IsTypedef, IsExtern, IsStatic, IsRegister, IsConst, IsVolatile: int;
```

This macro is defined in definitions 129, 130, and 131.
This macro is invoked in definition 4.

The `init_declarator_list_opt` is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared.

Semantics of declarations[131] ≡

```
SYMBOL declaration_specifiers: TypeSpecified: DefTableKey;
CHAIN                          TypeChain:     DefTableKey;

RULE: declaration ::= declaration_specifiers init_declarator_list_opt ';'
COMPUTE
  CHAINSTART init_declarator_list_opt.TypeChain=
      declaration_specifiers.TypeSpecified;
  init_declarator_list_opt.IsTypedef=declaration_specifiers.IsTypedef;
END;

SYMBOL init_declarator COMPUTE
  THIS.TypeChain=THIS.TypeChain DEPENDS_ON TAIL.TypeChain;
END;

RULE: init_declarator ::= declarator '=' initializer
COMPUTE
  IF(INCLUDING init_declarator_list_opt.IsTypedef,
     message(ERROR,"Cannot assign values to types", 0, COORDREF));
END;
```

This macro is defined in definitions 129, 130, and 131.
This macro is invoked in definition 4.

The `TypeSpecified` attribute embodies the type information indicated by the specifiers, and `TypeChain` carries that information to each declarator in the list. Additional type information provided by a declarator modifies the value of `TypeChain`, so each `init_declarator` in the list must reset the value for its successors.

6.5.1 Storage-class specifiers

Storage-class specifiers[132] ≡

```
storage_class_specifier:
  'typedef' A typedef storage class specifier has been accepted[217] /
  'extern' /
  'static' /
  'auto' /
  'register' .
```

This macro is invoked in definition 127.

Declaration_specifier equivalence class[133] ≡

```
declaration_specifier ::= storage_class_specifier .
```

This macro is defined in definitions 133, 135, and 152.
This macro is invoked in definition 3.

6.5.2 Type specifiers

Each list of type specifiers must consist of the specifiers making up one of the sets defined in the standard. Five of the 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 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 as `type_specifier_2`s.

Type specifiers[134] ≡

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

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

Structure and union specifiers[138]
Enumeration specifiers[142]

This macro is invoked in definition 127.

Declaration_specifier equivalence class[135] ≡

```
declaration_specifier ::= type_specifier_1 type_specifier_2 .
```

This macro is defined in definitions 133, 135, and 152.
This macro is invoked in definition 3.

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:

declaration_specifiers: [136] ≡

```
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 .
```

This macro is invoked in definition 127.

specifier_qualifier_list: [137] ≡

```
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 .
```

This macro is invoked in definition 138.

Structure and union specifiers *Structure and union specifiers*[138] ≡

```
struct_or_union_specifier:
  struct_or_union identifier component_list /
  struct_or_union component_list /
  struct_or_union identifier $';' .
```

component_list: [29]

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

struct_declaration_list:
  struct_declaration /
  struct_declaration_list struct_declaration.

struct_declaration:
  specifier_qualifier_list struct_declarator_list ';'.
```

```
specifier_qualifier_list: [137]

struct_declarator_list:
  struct_declarator /
  struct_declarator_list ',' struct_declarator.

struct_declarator:
  member_declarator /
  member_declarator_opt ':' constant_expression .

Member declarators[160]
```

This macro is invoked in definition 134.

A `member_declarator` must be distinguished from a `declarator` during parsing because the identifiers they declare belong to different name spaces, and those name spaces are treated differently as the program is parsed. A `member_declarator_opt` is semantically equivalent to a `member_declarator`:

Member_declarator equivalence class[139] ≡

```
member_declarator ::= member_declarator_opt .
```

This macro is invoked in definition 3.

A `component_list` defines the name space in which the members are defined. An instance of the Eli `Unique` module can be used to detect multiply defined members.

Name analysis modules for member identifiers[140] ≡

```
$/Prop/Unique.gnrc     +instance=Member +referto=Member :inst
$/Name/CScopeProp.gnrc +instance=Member +referto=Member :inst
```

This macro is invoked in definition 10.

The single name space is associated with the `file` phrase, and a separate environment is associated with each `component_list` phrase.

Semantics of structure and union specifiers[141] ≡

```
SYMBOL file           INHERITS MemberRootScope,  MemberRangeUnique    END;
SYMBOL component_list INHERITS MemberRangeScope, MemberRangeScopeProp END;

RULE: struct_declaration ::= declaration_specifiers struct_declarator_list ';'
COMPUTE
  CHAINSTART struct_declarator_list.TypeChain=
    declaration_specifiers.TypeSpecified;
END;

SYMBOL struct_declarator_list COMPUTE
  THIS.TypeChain=THIS.TypeChain DEPENDS_ON TAIL.TypeChain;
END;
/*      NOREPORTS
SYMBOL struct_declarator COMPUTE
  THIS.TypeChain=THIS.TypeChain DEPENDS_ON TAIL.TypeChain;
END;
```

```
*/

SYMBOL MemberIdDef INHERITS MemberIdDefScope, MemberUnique COMPUTE
  IF(NOT(THIS.MemberUnique),
    message(ERROR, "member identifier is multiply defined", 0, COORDREF));
END;

SYMBOL MemberIdUse:
  Sym:         int,
  MemberScope: Environment,
  MemberKey:   DefTableKey;

SYMBOL MemberIdUse COMPUTE
  SYNT.MemberKey=KeyInScope(THIS.MemberScope,THIS.Sym);
  IF(EQ(THIS.MemberScope,NoEnv),message(NOTE,"No environment",0,COORDREF));
  IF(EQ(THIS.MemberKey, NoKey),
    message(ERROR, "Not a member of this structure or union", 0, COORDREF));
END;
```

This macro is invoked in definition 4.

A `MemberIdUse` appears only as the right operand of a selection operator. The name space in which that member identifier was defined is the name space associated with the structure or union that is the left operand of that selection operator. Thus it is not possible to obtain a definition table key value at a `MemberIdUse` node until the analysis of expression types is complete.

Enumeration specifiers *Enumeration specifiers*[142] ≡

```
enum_specifier:
  'enum' identifier Begin a list of enumeration constants[218]
    '{' enumerator_list End a list of enumeration constants[219] '}' /
  'enum' Begin a list of enumeration constants[218]
    '{' enumerator_list End a list of enumeration constants[219] '}' /
  'enum' identifier .

enumerator_list:
  enumerator[25] /
  enumerator_list ',' enumerator[25] .

enumerator:
  enumeration_constant /
  enumeration_constant '=' constant_expression.
```

This macro is invoked in definition 134.

The identifiers in an enumerator list are declared as constants that have type `int` and may appear wherever such are permitted. Thus, the identifiers of enumeration constants declared in the same scope shall all be distinct from each other and from other identifiers declared in ordinary declarators.

Semantics of enumeration specifiers[143] ≡

```
SYMBOL enumeration_constant INHERITS Unique COMPUTE
  ResetType(THIS.Key, TypeIs_int);
  IF(NOT(THIS.Unique),
```

```
    message(ERROR, "identifier is multiply defined", 0, COORDREF));
END;
```

This macro is invoked in definition 4.

Tags A type specifier of the following form declares the identifier to be the tag of the structure, union or enumeration specified by the list:

Tags[144] ≡

```
SYMBOL TagDef: HasCurly: int;

RULE: struct_or_union_specifier ::= struct_or_union TagDef component_list
COMPUTE
  TagDef.HasCurly=1;
  struct_or_union_specifier.Type=
    KeyForStructOrUnion(TagDef.TagKey,struct_or_union.IsStruct)
    DEPENDS_ON struct_or_union_specifier._C_GotTypes;
  component_list.MemberScopeKey=struct_or_union_specifier.Type;
  component_list._C_GotTypes=struct_or_union_specifier.Type;
END;

RULE: enum_specifier ::= 'enum' TagDef '{' enumerator_list '}'
COMPUTE
  TagDef.HasCurly=1;
  enum_specifier.Type=
    KeyForEnum(TagDef.TagKey) DEPENDS_ON enum_specifier.Specification;
END;
```

This macro is invoked in definition 4.

If this declaration of the tag is visible, a subsequent declaration that uses the tag and that omits the bracketed list specifies the declared structure, union or enumerated type. Subsequent declarations in the same scope shall omit the bracketed list.

Check for multiply-declared tags[145] ≡

```
SYMBOL TagDef COMPUTE
  INH.HasCurly=0;
  THIS._C_TagGotKeys=
    IF(THIS.HasCurly,SetMultipleTag(THIS.TagKey,0,1))
    DEPENDS_ON THIS._C_TagGotKeys;
  IF(AND(THIS.HasCurly,GetMultipleTag(THIS.TagKey,0)),
    message(ERROR,"Multiply-defined tag",0,COORDREF))
    DEPENDS_ON INCLUDING file.TagGotKeys;
END;
```

This macro is invoked in definition 4.

Property characterizing multiply-declared tags[146] ≡

```
MultipleTag: int;
```

This macro is invoked in definition 9.

If a type specifier of the following form occurs prior to the declaration that defines the content, it declares a tag:

Tag that may or may not be a declaration[147] ≡

```
RULE: struct_or_union_specifier ::= struct_or_union identifier
COMPUTE
  .VisibleKey=
    KeyInEnv(INCLUDING TagAnyScope.TagEnv,identifier)
    DEPENDS_ON struct_or_union_specifier._C_TagGotKeys;
  .FinalKey=
    IF(NE(.VisibleKey,NoKey),
      .VisibleKey,
      DefineIdn(INCLUDING TagAnyScope.TagEnv,identifier));
  struct_or_union_specifier._C_TagGotKeys=.FinalKey;
  struct_or_union_specifier.Type=
    KeyForStructOrUnion(.FinalKey,struct_or_union.IsStruct)
    DEPENDS_ON struct_or_union_specifier._C_GotTypes;
  struct_or_union_specifier._C_GotTypes=struct_or_union_specifier.Type;
END;

ATTR VisibleKey, FinalKey: DefTableKey;
```

This macro is invoked in definition 4.

A similar construction with `enum` does not exist and is not necessary as there can be no mutual dependencies between the declaration of an enumerated type and any other type.

Use of an enumerated type tag[148] ≡

```
RULE: enum_specifier ::= 'enum' TagUse
COMPUTE
  enum_specifier.Type=
    GetType(TagUse.TagKey, NoKey) DEPENDS_ON enum_specifier.Specification;
END;

SYMBOL TagUse COMPUTE
  IF(EQ(THIS.TagKey, NoKey),
    message(ERROR, "Identifier is not defined", 0, COORDREF));
END;
```

This macro is invoked in definition 4.

A declaration of the following form specifies a structure or union type and declares a tag, both of which are visible only within the scope in which the declaration occurs:

Forward definition of a tag[149] ≡

```
RULE: declaration ::= struct_or_union TagDef ';'
COMPUTE
  declaration._C_GotTypes=
    KeyForStructOrUnion(TagDef.TagKey,struct_or_union.IsStruct)
    DEPENDS_ON declaration._C_GotTypes;
END;
```

This macro is invoked in definition 4.

A type specifier of the following form specifies a new structure, union or enumerated type, within the translation unit, that can only be referred to by the declaration of which it is a part.

Anonymous structure, union or enumerated type[150] ≡

```
RULE: struct_or_union_specifier ::= struct_or_union component_list
COMPUTE
  struct_or_union_specifier.Type=
    KeyForStructOrUnion(NoKey,struct_or_union.IsStruct)
    DEPENDS_ON struct_or_union_specifier._C_GotTypes;
  component_list.MemberScopeKey=struct_or_union_specifier.Type;
  component_list._C_GotTypes=struct_or_union_specifier.Type;
END;

RULE: enum_specifier ::= 'enum' '{' enumerator_list '}'
COMPUTE
  enum_specifier.Type=
    KeyForEnum(NoKey) DEPENDS_ON enum_specifier.Specification;
END;
```

This macro is invoked in definition 4.

6.5.3 Type qualifiers

Type qualifiers[151] ≡

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

This macro is invoked in definition 127.

Declaration_specifier equivalence class[152] ≡

```
declaration_specifier ::= type_qualifier .
```

This macro is defined in definitions 133, 135, and 152.
This macro is invoked in definition 3.

6.5.4 Declarators

Declarators[153] ≡

```
declarator:
  pointer Innermost declarator[26] Pointer declarator[197] /
  Innermost declarator[26] /
  pointer shielded Pointer declarator[197] /
  shielded .
```

This macro is defined in definitions 153, 155, 156, and 158.
This macro is invoked in definition 127.

The standard uses an optional `pointer`. This option must be made explicit here in order to avoid an LALR(1) conflict.

An ordinary identifier must be bound at the end of its declarator. If a declarator has another declarator as a component, then the identifier will be bound at the end of the component declarator and no binding should take place at the end of the outer declarator. A new nonterminal, `shielded`, is used to distinguish these cases so that the the parser can take the proper action. This distinction is only relevant during parsing; semantically, `shielded` is completely equivalent to `direct_declarator`:

Direct_declarator equivalence class[154] ≡

```
direct_declarator ::= shielded .
```

This macro is invoked in definition 3.

`OrdinaryIdDef` is a defining occurrence of an ordinary identifier. The kind of identifier being defined depends upon the context in which the `declarator` appears.

Declarators[155] ≡

```
direct_declarator:
  OrdinaryIdDef Defer binding the declared identifier[222] /
  direct_declarator Array declarator[196] '[' constant_exp_opt ']' /
  direct_declarator parameter_part .

shielded:
  '(' declarator ')' /
  shielded Array declarator[196] '[' constant_exp_opt ']' /
  shielded parameter_part .
```

This macro is defined in definitions 153, 155, 156, and 158.
This macro is invoked in definition 127.

The symbol `parameter_part` represents a significant semantic concept, the function prototype scope for tags. This phrase is not distinguished in the standard, making the corresponding semantics difficult to express.

Declarators[156] ≡

```
parameter_part:
  Prototype begin[186] '(' Begin a parameter list[198] parameters
    Prototype end[187] ')' .

parameters:
  Begin a parameter_type_list[194] parameter_type_list
    End a parameter_type_list[195] /
  identifier_list_opt .
```

This macro is defined in definitions 153, 155, 156, and 158.
This macro is invoked in definition 127.

The symbol `parameters` is introduced to simplify attachment of parsing actions, but it is semantically equivalent to a `parameter_part`:

Parameter_part equivalence class[157] ≡

```
  parameter_part ::= parameters .
```

This macro is invoked in definition 3.

Declarators[158] ≡

```
pointer:
```

```
    '*' type_qualifier_list_opt /
    '*' type_qualifier_list_opt pointer .

  type_qualifier_list_opt: type_qualifier* .

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

  parameter_list:
    parameter_declaration /
    parameter_list ',' parameter_declaration.

  parameter_declaration:
    declaration_specifiers declarator /
    declaration_specifiers abstract_declarator_opt .

  identifier_list:
    OrdinaryIdDef Bind the identifier immediately[224] /
    identifier_list ',' OrdinaryIdDef Bind the identifier immediately[224] .
```

This macro is defined in definitions 153, 155, 156, and 158.
This macro is invoked in definition 127.

Semantics of declarators[159] ≡

```
  RULE: direct_declarator ::= IdDef
  COMPUTE
    direct_declarator._C_GotTypes=
      ORDER(
        ResetType(IdDef.Key, direct_declarator.TypeChain),
        IF(NE(GetSynCode(IdDef.Key,UnboundIdUse),typedef_name),
          Can be the operand of &[162]('direct_declarator.TypeChain')),
        direct_declarator._C_GotTypes);
  END;

  RULE: member_direct_declarator ::= MemberIdDef
  COMPUTE
    member_direct_declarator._C_GotTypes=
      ORDER(
        Can be the operand of &[162]('member_direct_declarator.TypeChain'),
        ResetType(MemberIdDef.MemberKey,member_direct_declarator.TypeChain),
        member_direct_declarator._C_GotTypes);
  END;

  SYMBOL parameter_id COMPUTE
    THIS.TypeChain=ORDER(ResetType(THIS.Key, TypeIs_int),THIS.TypeChain);
  END;

  RULE: parameter_declaration ::= declaration_specifiers declarator
  COMPUTE
    CHAINSTART declarator.TypeChain=
      declaration_specifiers.TypeSpecified;
  END;
```

```
RULE: parameter_declaration ::= declaration_specifiers abstract_declarator
COMPUTE
  CHAINSTART abstract_declarator.TypeChain=
    declaration_specifiers.TypeSpecified;
  parameter_declaration._C_GotTypes=abstract_declarator.TypeChain;
END;
```

This macro is invoked in definition 4.

There is little difference between the phrase structure of a `member_declarator` and that of a `declarator`. The former is defined to preserve the context in which a declarator occurs, so that different actions can be attached in the two cases. A `declarator` declares an ordinary identifier, which must be consistently renamed during parsing, while a `member_declarator` declares a member identifier, which requires no action during parsing.

Member declarators[160] ≡

```
member_declarator:
  pointer member_direct_declarator / member_direct_declarator .

member_direct_declarator:
  identifier /
  '(' member_declarator ')' /
  member_direct_declarator '[' constant_exp_opt ']' /
  member_direct_declarator parameter_part .
```

This macro is invoked in definition 138.

Section 6.5.4.3 of the standard requires that an identifier list in a function declarator that is not part of a function definition be empty. Since a `member_declarator` cannot be a part of a function definition, there is no need for a non-empty identifier list alternative.

Pointer declarators *Pointer declarators*[161] ≡

```
RULE: pointer ::= '*' declaration_specifiers
COMPUTE
  pointer.TypeChain=
    KeyForQualifier(
      KeyForPointer(pointer.TypeChain),
      declaration_specifiers.IsConst,
      declaration_specifiers.IsVolatile);
END;

RULE: pointer ::=  '*' declaration_specifiers pointer
COMPUTE
  pointer[2].TypeChain=
    KeyForQualifier(
      KeyForPointer(pointer[1].TypeChain),
      declaration_specifiers.IsConst,
      declaration_specifiers.IsVolatile);
END;
```

This macro is invoked in definition 4.

The concrete syntax restricts the `declaration_specifiers` to be either empty or consist solely of type qualifiers. `KeyForQualifier` accounts for any qualifiers that might be present.

Pointer types also have to be guaranteed whenever an object that could be the operand of `&` is declared. The reason is that "referencing" operators are defined for each pointer type, so the type must be instantiated before `&` can be identified.

Can be the operand of &[162](◇1) ≡

```
KeyForPointer(◇1)
```
This macro is invoked in definition 159.

Array declarators *Array declarators*[163] ≡

```
RULE: direct_declarator ::= direct_declarator '[' constant_exp_opt ']'
COMPUTE
  direct_declarator[2].TypeChain=KeyForArray(direct_declarator[1].TypeChain);
END;

RULE: member_direct_declarator ::=
  member_direct_declarator '[' constant_exp_opt ']'
COMPUTE
  member_direct_declarator[2].TypeChain=
    KeyForArray(member_direct_declarator[1].TypeChain);
END;

RULE: direct_abs_declarator ::= '[' constant_exp_opt ']'
COMPUTE
  direct_abs_declarator.TypeChain=KeyForArray(direct_abs_declarator.TypeChain);
END;

RULE: direct_abs_declarator ::= direct_abs_declarator '[' constant_exp_opt ']'
COMPUTE
  direct_abs_declarator[2].TypeChain=
    KeyForArray(direct_abs_declarator[1].TypeChain);
END;
```

This macro is invoked in definition 4.

Function declarators (including prototypes) *Function declarators (including prototypes)*[164] ≡

```
RULE: direct_declarator ::= direct_declarator parameter_part
COMPUTE
  direct_declarator[2].TypeChain=
    KeyForFunction(direct_declarator[1].TypeChain);
END;

RULE: member_direct_declarator ::= member_direct_declarator parameter_part
COMPUTE
  member_direct_declarator[2].TypeChain=
    KeyForFunction(member_direct_declarator[1].TypeChain);
END;

RULE: direct_abs_declarator ::= '(' parameter_type_list_opt ')'
COMPUTE
  direct_abs_declarator.TypeChain=
```

```
    KeyForFunction(direct_abs_declarator.TypeChain);
END;

RULE: direct_abs_declarator ::=
     direct_abs_declarator '(' parameter_type_list_opt ')'
COMPUTE
  direct_abs_declarator[2].TypeChain=
    KeyForFunction(direct_abs_declarator[1].TypeChain);
END;
```

This macro is invoked in definition 4.

6.5.5 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 an LALR(1) conflict.

Type names[165] ≡

```
type_name:
  specifier_qualifier_list abstract_declarator_opt .

abstract_declarator:
  pointer /
  pointer direct_abs_declarator / direct_abs_declarator .

direct_abs_declarator:
  '(' abstract_declarator ')' /
  direct_abs_declarator '[' constant_exp_opt ']' /
  direct_abs_declarator '(' parameter_type_list_opt ')' /
  '[' constant_exp_opt ']' /
  '(' parameter_type_list_opt ')' .
```

This macro is invoked in definition 127.

An `abstract_declarator_opt` is sematically equivalent to an `abstract_declarator`:

Abstract_declarator equivalence class[166] ≡

```
abstract_declarator ::= abstract_declarator_opt .
declaration_specifiers ::= type_qualifier_list_opt specifier_qualifier_list .
```

This macro is invoked in definition 3.

Semantics of type names[167] ≡

```
RULE: type_name ::= declaration_specifiers abstract_declarator
COMPUTE
  CHAINSTART abstract_declarator.TypeChain=
    declaration_specifiers.TypeSpecified;
  type_name.Type=abstract_declarator.TypeChain;
  type_name._C_GotTypes=type_name.Type;
END;
```

This macro is invoked in definition 4.

6.5.6 Type definitions

In order to avoid syntactic ambiguity, a `typedef_name` must be recognized by the lexical analyzer.

Type definitions[168] ≡

```
typedef_name:  $[_a-zA-Z][_a-zA-Z0-9]*
```

This macro is invoked in definition 16.

Note that there is no lexical distinction between a `typedef_name` and an `identifier`. Thus semantic information must be used to differentiate them, and the differentiation must occur during parsing.

6.5.7 Initialization

Initialization[169] ≡

```
initializer:
  assignment_expression /
  '{' initializer_list '}' /
  '{' initializer_list ',' '}'.

initializer_list:
  initializer /
  initializer_list ',' initializer.
```

This macro is invoked in definition 127.

6.6 Statements

Statements[170] ≡

```
statement:
  labeled_statement /
  Begin a nested compound statement[189] compound_statement /
  expression_statement /
  selection_statement /
  iteration_statement /
  jump_statement.

Labeled statements[171]
Compound statements[172]
Expression statements[173]
Selection statements[174]
Iteration statements[175]
Jump statements[176]
```

This macro is invoked in definition 2.

6.6.1 Labeled statements

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

In the absence of parser information, the lexical analyzer will classify an identifier as an ordinary identifier (one of `UnboundIdUse`, `OrdinaryIdUse` or `typedef_name` depending on the context). Thus the parser must accept any of these classifications as a valid label.

Labeled statements[171] ≡

```
labeled_statement:
  UnboundIdUse ':' statement /
  OrdinaryIdUse ':' statement /
  typedef_name ':' statement /
  'case' constant_expression ':' statement /
  'default' ':' statement.
```

This macro is invoked in definition 170.

6.6.2 Compound statements

Compound statements[172] ≡

```
compound_statement:
  '{' declaration_list statement_list_opt End a compound statement[188] '}' /
  '{' statement_list_opt End a compound statement[188] '}' .

declaration_list:
  declaration /
  declaration_list declaration .

statement_list:
  statement /
  statement_list statement .
```

This macro is invoked in definition 170.

6.6.3 Expression and null statements

Expression statements[173] ≡

```
expression_statement: expression_opt ';' .
```

This macro is invoked in definition 170.

6.6.4 Selection statements

There is an LALR(1) 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`.

Selection statements[174] ≡

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

This macro is invoked in definition 170.

6.6.5 Iteration statements

The nointerminals `for_init`, `for_test` and `for_incr` are all optional expressions. They are used to preserve context without a combinatorial explosion.

Iteration statements[175] ≡

```
iteration_statement:
  'while' '(' expression ')' statement /
  'do' statement 'while' '(' expression ')' ';' /
  'for' '(' for_init ';' for_test ';' for_incr ')' statement.
for_init: / expression .
for_test: / expression .
for_incr: / expression .
```

This macro is invoked in definition 170.

6.6.6 Jump statements

Jump statements[176] ≡

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

This macro is invoked in definition 170.

6.7 External definitions

External definitions[177] ≡

```
translation_unit:
  external_declaration /
  translation_unit external_declaration .

external_declaration:
  BeginExtDecl function_definition /
  BeginExtDecl declaration .

BeginExtDecl: Begin an external declaration[191] .
```

This macro is invoked in definition 2.

6.7.1 Function definitions

The standard uses an optional declaration_specifiers. This option must be made explicit here in order to avoid an LALR(1) conflict.

Function definitions[178] ≡

```
function_definition:
  declaration_specifiers declarator Function definition[190]
    declaration_list_opt compound_statement /
  declarator Function definition[190]
    declaration_list_opt compound_statement .
```

This macro is invoked in definition 2.

Semantics of function definitions[179] ≡

```
RULE: function_definition ::=
     declaration_specifiers declarator declaration_list_opt compound_statement
COMPUTE
  CHAINSTART declarator.TypeChain=declaration_specifiers.TypeSpecified;
END;

RULE: function_definition ::=
                              declarator declaration_list_opt compound_statement
COMPUTE
  CHAINSTART declarator.TypeChain=TypeIs_int;
END;
```

This macro is invoked in definition 4.

6.7.2 External object definitions

7 Library

8 Consistent Renaming for Ordinary Identifiers

An ordinary identifier may be accepted by the parser as any of the terminals OrdinaryIdDef, OrdinaryIdUse, typedef_name, UnboundIdUse or identifier. Defining occurrences are represented in the tree by either an IdDef node or a parameter_id node, applied occurrences by an IdUse node.

If the parser has accepted an OrdinaryIdUse or a typedef_name in the context of a LabelDef, then that identifier must be renamed in the label name space. The original identifier was stored as the Symbol property of the ordinary identifier, so it can be recovered easily:

Semantics of identifiers[180] ≡

```
RULE: LabelDef ::= OrdinaryIdUse
COMPUTE
   LabelDef.Sym = GetSymbol(OrdinaryIdStackArray(OrdinaryIdUse), 0);
END;
```

```
RULE: LabelDef ::= typedef_name
COMPUTE
   LabelDef.Sym = GetSymbol(OrdinaryIdStackArray(typedef_name), 0);
END;

RULE: LabelDef ::= UnboundIdUse
COMPUTE
   LabelDef.Sym = UnboundIdUse;
END;

RULE: LabelUse ::= identifier
COMPUTE
   LabelUse.Sym = identifier;
END;

RULE: TagDef ::= identifier
COMPUTE
  TagDef.Sym=identifier;
END;

RULE: TagUse ::= identifier
COMPUTE
  TagUse.Sym=identifier;
END;

RULE: MemberIdDef ::= identifier
COMPUTE
  MemberIdDef.Sym = identifier;
END;

RULE: MemberIdUse ::= identifier
COMPUTE
  MemberIdUse.Sym = identifier;
END;

RULE: enumeration_constant ::= OrdinaryIdDef
COMPUTE
  enumeration_constant.Key=OrdinaryIdStackArray(OrdinaryIdDef);
END;
```

This macro is invoked in definition 4.

Scope rules are embodied in the concept of an *environment*, which is a set of bindings for identifiers. Environments can be nested, in which case the binding in the innermost environment hides those in outer environments. Eli provides a module that exports an `Environment` data type with operations to establish bindings, find bindings and create nested sets of bindings. This module is used to support the consistent renaming process.

Ordinary identifier name space[181] ≡

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

This macro is invoked in definition 27.

The environment module provides two operations for creating environments: `NewEnv` and `NewScope`. `NewEnv` should be used once to create the `Environment` value for the outermost scope of a name space, and `NewScope` should be used to create `Environment` values for nested scopes. An `Environment` value represents a set of bindings, and may be related to another `Environment` value.

The operations that create bindings (`DefineIdn` and `AddIdn`) take an `Environment` value as a parameter. If no element of the set is a binding for the given identifier, they add a binding to the set; otherwise the set is left unchanged. In either case, the set will contain a binding for the given identifier after the operation is completed. `DefineIdn` returns the definition table key bound to the identifier, while `AddIdn` returns an integer value that is 1 if the specified binding was added to the set and 0 otherwise.

The operations that query bindings (`KeyInScope` and `KeyInEnv`) also take an `Environment` value as a parameter. `KeyInScope` returns the definition table key bound to the identifier in the set, if there is one, and returns the distinguished definition table key `NoKey` if the set contains no binding for the identifier. `KeyInEnv` behaves the same way except that if there is no binding in the given set then it examines the related sets as well. `NoKey` is returned only if there is no binding for the identifier in *any* related set.

Consistent renaming for ordinary identifiers must be done on the fly as the text is read, in order to allow the lexical analyzer to recognize certain character sequences as `typedef_name`s. Labels, tags and members need not be distinguished lexically, and therefore consistent renaming in these name spaces can be deferred until after the tree is built.

In order to carry out the consistent renaming process as the input text is read, the environment module's operations must be invoked during lexical analysis and parsing. These invocations must be specified as part of the descriptions of the lexical analysis and parsing problems for C, using *token processors* and *connections* respectively. (A token processor is an arbitrary C routine that is invoked after a specified character sequence has been recognized. The name of the routine, enclosed in square brackets, is added to the specification of the character sequence. A connection is arbitrary C code that is executed when the parser reduces a specified production. The code, enclosed in apostrophes and preceded by an ampersand, is added to the specification of the production.)

The keys that were assigned to the ordinary identifiers during parsing are not stored as values of the `OrdinaryIdDef`, `OrdinaryIdUse`, `typedef_name` and `UnboundIdUse` terminals because Eli requires that a parsed terminal have an integer value. Instead, the keys are stored in array `OrdinaryIdStackArray` and the index of the array element becomes the value of the terminal. An array access is therefore needed to obtain the key:

Access to the key attribute for ordinary identifiers[182] $\equiv$

```
ATTR Key: DefTableKey;

RULE: IdDef ::= OrdinaryIdDef
COMPUTE
  IdDef.Key = OrdinaryIdStackArray(OrdinaryIdDef);
END;

RULE: IdUse ::= OrdinaryIdUse
COMPUTE
  IdUse.Key = OrdinaryIdStackArray(OrdinaryIdUse);
END;

RULE: IdUse ::= UnboundIdUse
COMPUTE
  IdUse.Key = NewKey();
  IF(NOT(IdUse.FunctionId),
    message(ERROR,"Undefined identifier",0,COORDREF));
END;

ATTR FunctionId: int;
```

```
RULE: Expression ::= IdUse
COMPUTE
  IdUse.FunctionId=Expression.FunctionId;
END;

RULE: Expression ::= Expression '(' argument_exp_list ')'
COMPUTE
  Expression[2].FunctionId=1;
END;

RULE: Expression ::= Expression '(' ')'
COMPUTE
  Expression[2].FunctionId=1;
END;

SYMBOL Expression COMPUTE INH.FunctionId=0; END;

RULE: identifier_list LISTOF parameter_id END;
RULE: parameter_id ::= OrdinaryIdDef
COMPUTE
  parameter_id.Key = OrdinaryIdStackArray(OrdinaryIdDef);
END;
```

This macro is invoked in definition 4.

There are two preconditions for this computation: the `OrdinaryIdentifier` must have a value, and the element of `OrdinaryIdStackArray` indexed by that value must contain the appropriate key. According to the scope rules of C, the first of these preconditions can be true and the second false.

Both preconditions are guaranteed to hold at the time the tree construction is complete. Therefore a sufficient condition for correctness of the computation is to delay it until that time:

Do not allow attribution during tree construction[183] ≡

```
ORDER: TREE COMPLETE;
```

This macro is invoked in definition 13.

Within the scope of a particular identifier, all occurrences of that identifier have the same meaning. The meaning of an identifier is embodied in a number of properties, which vary depending on the kind of identifier. For example, a variable identifier might have properties like the type of the variable and its memory address; an enumeration constant would have a value.

Eli provides a central data base called the *definition table*, in which arbitrary properties can be stored. The properties of a given entity are queried or updated via a *definition table key* for that entity. Thus it is reasonable to represent an identifier by a definition table key, and store that definition table key in the tree at each occurrence of the identifier. Then it is possible to query or update the properties of the identifier during computations over the tree.

Consistent renaming is the process of associating definition table keys with occurrences of identifiers: The identifier is "renamed" by the definition table key, and the process is "consistent" because all of the occurrences of an identifier within its scope are associated with the same definition table key.

This section defines the code needed to perform consistent renaming in the ordinary identifier name space of C.

To implement the scope rules for ordinary identifiers, we provide an `Environment` value for each set of identifiers that have the same scope. At the beginning of the scope of an identifier, the parser will add a binding for that identifier to the current environment. At the common end of the scopes for identifiers bound in the current environment, the `Environment` value will be discarded.

Identifier scopes are nested, as indicated in Section 1.1.2.1. This means that a stack can be used to store the `Environment` values. Any operation that creates or queries a binding will use the top element of that stack as its `Environment` parameter. The easiest way to obtain such a stack is to create an instance of the generic stack module that is specialized to store `Environment` values:

Create a module to stack Environment values[184] $\equiv$

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

This macro is invoked in definition 10.

Because the `instance` parameter value is `Region`, the operations exported by this instance of the stack module will be `RegionStackPush`, `RegionStackPop`, `RegionStackTop`, and so forth.

The remaining subsections of this section explain how the three scopes defined by the standard are implemented in this specification.

8.1 File scope

The `Environment` value for all identifiers with file scope must be created and stored in `RegionStack` before parsing begins, because an erroneous program might begin with an ordinary identifier and the environment must be available for the classification algorithm to run:

Initialize the set of bindings having file scope[185] $\equiv$

```
RegionStackPush(NewEnv());
```

This macro is invoked in definition 6.

There is no need to explicitly discard this value, since parsing ends at the point where the value would be discarded. Thus there will be no references to *any* `Environment` value beyond that point.

8.2 Function prototype scope

According to Section 1.1.2.1, a function prototype scope ends at the end of the function declarator. Since two identifiers have the same scope if and only if their scopes terminate at the same point, bindings for all parameter identifiers in a given declarator should be in the same set and that set should be associated with the declarator. Without loss of error detection capability, however, it is possible to associate the set with a single parameter list. This fact greatly simplifies the implementation.

8.2.1 Constraints on function declarators

Section 6.5.4.3 of the standard states that "a function declarator shall not specify a return type that is a function type or an array type." Further, it gives the meaning of a declarator containing a parameter list by postulating a declaration `T D1` in which `D1` has the form `D(parameter_type_list)` or `D(identifier_list_opt)`. If the type of `T D` would be "`some_type T`" then the type of `T D1` is "`some_type` function returning `T`".

Now suppose that `D` has the same form as `D1`. Then the type of `T D` would be "`another_type` function returning `T`", so the type of `T D1` would be "`another_type` function returning function returning `T`". Since a function cannot return a function, this type is illegal. Therefore we must conclude that `D` cannot have the supposed form.

Section 6.5.4.2 of the standard gives the meaning of a declarator containing an array bounds expression by postulating a declaration `T D1` in which `D1` has the form `D[constant_exp_opt]`. If the type of `T D` would be "`some_type T`" then the type of `T D1` is "`some_type` array of `T`".

Now suppose that `D` has the form `D(parameter_type_list)` or `D(identifier_list_opt)`. Then the type of `T D` would be "`another_type` function returning `T`", so the type of `T D1` would be "`another_type` function returning array of `T`". Since a function cannot return an array, this type is illegal. Therefore we must conclude that `D` cannot have the supposed form.

But if a parameter list can be followed by neither another parameter list nor an expression in brackets, then it must be the last element of its declarator according to the grammar. Therefore the end of the parameter list must coincide with the end of the declarator. Finally, if the end of a parameter list coincides with the end of the declarator then a declarator cannot contain more than one parameter list.

These constraints are not imposed by the grammar, but they can easily be checked as the type of an identifier is determined. The checks depend only on the phrase structure and would not be invalidated by any error in consistent renaming. Thus it is safe to consider a function prototype scope to consist only of a single parameter list.

8.2.2 Implementation of function prototype scopes

Since the function prototype scope can be considered to consist only of a single parameter list, all that is necessary is to push a new environment at the beginning of the list and discard it at the end. The `Environment` value must represent a scope nested in the scope surrounding the parameter list:

Prototype begin[186] ≡

```
&'RegionStackPush(NewScope(RegionStackTop));'
```

This macro is invoked in definitions 14 and 156.

Prototype end[187] ≡

```
&'RegionStackPop;'
```

This macro is invoked in definitions 14 and 156.

8.3 Block scope

The ordinary identifiers that have block scope are those whose declarators appear either inside the block or within the list of parameter declarations in a function definition. An `Environment` value for all identifiers in a block scope is therefore created either at the beginning of the block (if it isn't the block of a function definition), or at the beginning of the function's parameter list. In either case, the value is discarded at the end of the block:

End a compound statement[188] ≡

```
&'(void)RegionStackPop;'
```

This macro is invoked in definition 172.

A block that is not part of a function definition gets a new `Environment` value that represents a scope nested in the surrounding scope:

Begin a nested compound statement[189] ≡

```
&'RegionStackPush(NewScope(RegionStackTop));'
```

This macro is invoked in definition 170.

If the block is part of a function definition, then it must use the `Environment` value that was used for the function prototype scope of the function's parameters. If the program is syntactically incorrect, however, there may be no such Environment value and a new nested scope must be created. Determination of the appropriate Environment value for a function body is a rather complex process, and therefore we embed it in a module:

Function definition[190] ≡

```
&'RegionStackPush(FunctionEnv(RegionStackTop));'
```

This macro is invoked in definition 178.

8.3.1 Determining the function environment

The syntactic structure of a C function definition does not reflect the scope rules. A function definition may contain an arbitrary number of parameter_parts, some embedded in type specifiers and others in the declarator. The specific parameter_part declaring ordinary identifiers to be parameters of the function can be determined from the tree structure, but the tree structure is not available during parsing. Therefore the parser must use an alternative method, based only on left context, to select the appropriate parameter_part.

A function_definition is an external_declaration. Therefore the left context of interest does not extend beyond the beginning of an external declaration:

Begin an external declaration[191] ≡

```
&'NewDeclaration();'
```

This macro is invoked in definition 177.

Function declarators always appear at the "top level" of an external_declaration, and hence a declarator appearing within a component_list or a parameter_type_list is uninteresting in this computation:

Begin a component_list[192] ≡

```
&'BeginNestedDecl();'
```

This macro is invoked in definition 29.

End a component_list[193] ≡

```
&'EndNestedDecl();'
```

This macro is invoked in definition 29.

Begin a parameter_type_list[194] ≡

```
&'BeginNestedDecl();'
```

This macro is invoked in definitions 14 and 156.

End a parameter_type_list[195] ≡

```
&'EndNestedDecl();'
```

This macro is invoked in definitions 14 and 156.

If a declarator specifies an array type to the left of the first specification of a function type, then that declarator is not a function declarator:

Array declarator[196] ≡

```
&'ArrayOrPointerDecl();'
```

This macro is invoked in definition 155.

Similarly, if a declarator specifies a pointer then no enclosing declarator is a function declarator:

Pointer declarator[197] ≡

```
&'ArrayOrPointerDecl();'
```

This macro is invoked in definition 153.

Depending upon the context, any parameter list might be the parameter list of a function definition. If so, then the `Environment` value for that parameter list will be the `Environment` value for the function body:

Begin a parameter list[198] ≡

```
&'FunctionDecl(RegionStackTop);'
```

This macro is invoked in definition 156.

8.3.2 Implementing the declaration state

The heart of the declaration state computation is a variable that distinguishes four possible states:

State variable definitions[199] ≡

```
typedef enum {
  UnknownDecl,
  IsFunction,
  NotFunction,
  IsNested
} DeclarationStateValues;

static int DeclarationState;
```

This macro is defined in definitions 199, 200, 201, 202, 203, 207, and 213.
This macro is invoked in definition 6.

Arbitrary nesting depth can be 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`.

State variable definitions[200] ≡

```
void BeginNestedDecl() { DeclarationState += IsNested; }
void EndNestedDecl() { DeclarationState -= IsNested; }
```

This macro is defined in definitions 199, 200, 201, 202, 203, 207, and 213.
This macro is invoked in definition 6.

The state is reset at the beginning of each external definition:

State variable definitions[201] ≡

```
void NewDeclaration() { DeclarationState = UnknownDecl; }
```

This macro is defined in definitions 199, 200, 201, 202, 203, 207, and 213.
This macro is invoked in definition 6.

When the parser reaches a specification of a pointer, array or function it can decide whether it is dealing with a function definition or not. That decision is made on the basis of the *leftmost* such specification that is at the top level. In other words, a decision can only be made in the `UnknownDecl` state:

State variable definitions[202] ≡

```
void ArrayOrPointerDecl()
{ if (DeclarationState == UnknownDecl) DeclarationState = NotFunction; }

static Environment FunctionEnvironment;

void
#if PROTO_OK
FunctionDecl(Environment Env)
#else
FunctionDecl(Env) Environment Env;
#endif
{ if (DeclarationState == UnknownDecl) {
    DeclarationState = IsFunction;
    FunctionEnvironment = Env;
  }
}
```

This macro is defined in definitions 199, 200, 201, 202, 203, 207, and 213.
This macro is invoked in definition 6.

If the state is `IsFunction` when the function body is reached, then `FunctionEnvironment` is the appropriate `Environment` value. Otherwise a new value representing a scope nested in the current scope must be created:

State variable definitions[203] ≡

```
Environment
#if PROTO_OK
FunctionEnv(Environment Env)
#else
FunctionEnv(Env) Environment Env;
#endif
{ if (DeclarationState == IsFunction) return FunctionEnvironment;
  return NewScope(Env);
}
```

This macro is defined in definitions 199, 200, 201, 202, 203, 207, and 213.
This macro is invoked in definition 6.

The interface to this module must include the interface to the environment module:

State variable declarations[204] ≡

```
#include "envmod.h"

extern void BeginNestedDecl ELI_ARG((void));
extern void EndNestedDecl ELI_ARG((void));
```

```
extern void NewDeclaration ELI_ARG((void));
extern void FunctionDecl ELI_ARG((Environment Env));
extern Environment FunctionEnv ELI_ARG((Environment Env));
```

This macro is defined in definitions 204, 208, and 212.
This macro is invoked in definition 7.

8.4 Occurrences of ordinary identifiers

An applied occurrence of an ordinary identifier that has been declared as the name of a type is represented in the grammar by the terminal symbol `typedef_name`. It is necessary to distinguish other applied occurrences of ordinary identifiers from occurrences of labels, tags and member identifiers because the consistent renaming process has been carried out for ordinary identifiers during parsing, and therefore an ordinary identifier has an associated definition table key. Labels, tags and members do not have definition table keys until the tree computation implementing the consistent renaming process for their respective name spaces has been completed. Therefore an applied occurrence of an ordinary identifier that has been declared as a variable, function, parameter or enumeration constant is represented in the grammar by the terminal symbol `OrdinaryIdUse`.

The lexical analyzer actions needed at the defining occurrences of ordinary identifiers are different from those needed at applied ocurrences. Thus the lexical analyzer must be able to determine when it is dealing with a defining occurrence of an ordinary identifier. If we represent the defining occurrence of an ordinary identifier in the grammar by yet another terminal symbol, `OrdinaryIdDef`, then the lexical analyzer will know that it is dealing with a defining occurrence of an ordinary identifier if the parser will accept the symbol `OrdinaryIdDef`.

An identifier can be used as the name of an external function even though that identifier has not been declared. Such an identifier can occur only in an expression context, and that context must also be represented by a special terminal symbol, `UnboundIdUse`, to provide the necessary information to the lexical analyzer.

Thus this specification defines three terminal symbols for identifiers in addition to the two (`identifier` and `typedef_name`) used by the standard:

Additional terminal symbols representing identifiers[205] ≡

```
OrdinaryIdUse:  $[_a-zA-Z][_a-zA-Z0-9]*
OrdinaryIdDef:  $[_a-zA-Z][_a-zA-Z0-9]*
UnboundIdUse:   $[_a-zA-Z][_a-zA-Z0-9]*
```

This macro is invoked in definition 16.

8.4.1 Ordinary identifiers and the definition table

Because Eli allows only integers as values for parsed terminals, it is not possible to represent ordinary identifiers by definition table keys. Instead, the specification defines an array of definition table keys, with one element for each defining occurrence and one element for all undefined ordinary identifiers. The representation of an ordinary identifier leaf is the index of the element for the corresponding defining occurrence.

Since the size of the array cannot be known a priori, it is implemented by a stack:

Create a module to stack DefTableKey values[206] ≡

```
$/Adt/Stack.gnrc +instance=OrdinaryId +referto=DefTableKey :inst
```

This macro is invoked in definition 10.

A state variable keeps track of the index of the next element to be allocated on the stack, which is initialized with one element specifying the distinguished definition table key `NoKey`. `NoKey` is the definition table key assigned to undefined identifiers:

State variable definitions[207] ≡

```
int NextKeyIndex = 1;
```

This macro is defined in definitions 199, 200, 201, 202, 203, 207, and 213.
This macro is invoked in definition 6.

State variable declarations[208] ≡

```
extern int NextKeyIndex;
```

This macro is defined in definitions 204, 208, and 212.
This macro is invoked in definition 7.

Initialize the array of definition table keys[209] ≡

```
  OrdinaryIdStackPush(NoKey);
```

This macro is invoked in definition 6.

The value of `NextKeyIndex` at any time is the number of elements in `OrdinaryIdStack`. In order to maintain this invariant, two actions are needed whenever a new key is allocated:

New key[210] ≡
`IdState.Index=NextKeyIndex++; OrdinaryIdStackPush(NoKey);`This macro is invoked in definitions 222 and 224.

Each defined ordinary identifier has three properties that must be established during parsing: a *symbol*, a *syntax code* and an *index*. The symbol is the index of the identifier's string in the character string memory array. Each string is stored exactly once, so the the values of the symbols for two identifiers with the same spelling will be the same. The syntax code is the integer classification of the identifier, and is always either `typedef_name` or `OrdinaryIdUse`. The index is the index of the definition table key in the array of defining occurrences.

Properties of ordinary identifiers used during parsing[211] ≡

```
Symbol, SynCode, Index: int;
```

This macro is invoked in definition 9.

8.4.2 State information for an ordinary identifier definition

In order to correctly define an ordinary identifier, state information must be retained to relate actions by the lexical analyzer and actions by the parser. The necessary information is exactly the set of properties for the identifier:

State variable declarations[212] ≡

```
typedef struct {
  int Symbol, SynCode, Index;
} IdProperties;

extern IdProperties IdState;
```

This macro is defined in definitions 204, 208, and 212.
This macro is invoked in definition 7.

State variable definitions[213] ≡

```
IdProperties IdState;
```

This macro is defined in definitions 199, 200, 201, 202, 203, 207, and 213.
This macro is invoked in definition 6.

`IdState` is a global variable that reflects the state information for the last defining occurrence of an ordinary identifier. The standard specifies (Section 6.1.2.1) that a defining occurrence is bound to a definition table key at the end of the declarator in which it appears (unless it is an `enumeration_constant`). An arbitrary number of other defining occurrences may appear between a particular defining occurrence and the point at which it is bound. Therefore a stack must be used to save `IdState` until the identifier it describes can be bound:

Create a module to stack identifier state values[214] ≡

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

This macro is invoked in definition 10.

The value of `IdState.Symbol` is set by the lexical analyzer for each occurrence of an identifier, and that of `IdState.Index` is set by the parser when it has accepted a defining occurrence of an ordinary identifier. `IdState.SynCode` is initialized to `OrdinaryIdUse`, and reset to that value by the parser whenever the end of a `declaration` is reached:

Initialize the class of ordinary identifiers[215] ≡

```
IdState.SynCode = OrdinaryIdUse;
```

This macro is invoked in definition 12.

End of a declaration[216] ≡

```
&'IdState.SynCode = OrdinaryIdUse;'
```

This macro is invoked in definition 127.

When the parser accepts the storage class specifier `typedef`, it sets `IdState.SynCode` to `typedef_name`. This setting will persist until the end of the `declaration` containing the `typedef`:

A typedef storage class specifier has been accepted[217] ≡

```
&'IdState.SynCode = typedef_name;'
```

This macro is invoked in definition 132.

An `enumeration_constant` is never a `typedef_name`, even when the enumeration specifier follows a typedef `storage_class_specifier`:

Begin a list of enumeration constants[218] ≡

```
&'IdStateStackPush(IdState); IdState.SynCode = OrdinaryIdUse;'
```

This macro is invoked in definition 142.

End a list of enumeration constants[219] ≡

```
&'IdState = IdStateStackPop;'
```

This macro is invoked in definition 142.

8.4.3 Classifying occurrences of identifiers

The lexical analyzer must classify each character sequence that matches the definition of `identifier` as one of the five terminal symbols `identifier`, `typedef_name`, `OrdinaryIdUse`, `OrdinaryIdDef` or `UnboundIdUse`. It does this by means of the token processor `IdnOrType`:

Initial classification of identifier terminals[220] ≡

```
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
/* Obtain the internal coding 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
 *    On exit-
 *       syncode has been set to the terminal code
 *       rep has been set to the internal coding
 ***/
{ DefTableKey key;

  mkidn(start, length, syncode, &(IdState.Symbol));
  key = KeyInEnv(RegionStackTop, IdState.Symbol);
  *syncode = GetSynCode(key, UnboundIdUse);
  *rep = GetIndex(key, IdState.Symbol);
}
```

This macro is invoked in definition 6.

`IdnOrType` first uses `mkidn` to obtain the index of the character sequence in the character string memory array, placing that index into the state variable component `IdState.Symbol`. Next, the index is looked up in the `RegionStackTop` environment to obtain the corresponding definition table key (if one exists). The `SynCode` and `Index` properties of that definition table key are then queried, and the results of the queries become the classification of the terminal symbol and the representation respectively. Since `UnboundIdUse` is specified as the default value for the `SynCode` query, the terminal symbol will be classified as an `UnboundIdUse` unless a previous binding is visible in the current environment. Similarly, if there is no visible binding, the representation will be set to the value of the state variable `NextKeyIndex`.

The classification determined by `IdnOrType` may or may not fit the current context. If it does fit, then the parser will accept the terminal symbol and continue. If it does not fit, the parser will invoke the function `Reparatur`:

Re-classify an identifier terminal to fit the context[221] ≡

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

```
#endif
/* 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 ||
      *syncode == OrdinaryIdUse ||
      *syncode == UnboundIdUse) {
    *syncode = OrdinaryIdDef;
    *rep = NextKeyIndex;
    return 1;
  }
  if (*syncode != OrdinaryIdDef) return 0;
  *syncode = identifier;
  *rep = IdState.Symbol;
  return 1;
}
```

This macro is invoked in definition 6.

If the original classification was based on a visible definition, or if the identifier had not been previously classified, then `Reparatur` re-classifies the symbol as a defining occurrence of an ordinary identifier. Again, the parser will either accept the terminal symbol with this classification or invoke `Reparatur`. If `Reparatur` is invoked with the previous classification being a defining occurrence of an ordinary identifier, then it re-classifies the symbol as an identifier in one of the other name spaces (label, tag or member).

8.4.4 Establishing bindings for ordinary identifiers

In order to establish the binding for an ordinary identifier, the parser must complete the identifier's state when the defining occurrence has been accepted or when the identifier has been accepted as unbound.

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.

Defer binding the declared identifier[222] ≡

&'*New key*[210] `IdStateStackPush(IdState);'`

This macro is invoked in definitions 51 and 155.

Carry out a deferred binding[223] ≡

```
&'IdState = IdStateStackPop; Bind();'
```

This macro is invoked in definitions 25 and 26.

Bind the identifier immediately[224] ≡

```
&'New key[210] Bind();'
```

This macro is invoked in definition 158.

In either case, the actual binding process is embedded in a function because of its relative complexity:

Bind a defining occurrence of an ordinary identifier[225] ≡

```
void
Bind()
{ DefTableKey key;

  key = DefineIdn(RegionStackTop, IdState.Symbol);
  ResetSymbol(key, IdState.Symbol);
  ResetSynCode(key, IdState.SynCode);
  ResetIndex(key, IdState.Index);
  OrdinaryIdStackArray(IdState.Index) = key;
}
```

This macro is invoked in definition 6.

Operation interfaces[226] ≡

```
extern void Bind();
```

This macro is defined in definitions 226.
This macro is invoked in definition 7.

9 Internal Representation of Types

Each distinct type is represented internally by a distinct definition table key. Relationships among types are represented by `DefTableKey`-valued properties. Other characteristics, such as whether or not the type is complete, are also represented by appropriate properties. This chapter describes the structure of the internal representation, and the operations used to maintain it.

9.1 Creating type representations

The generated processor maintains a run-time data structure separate from the tree to represent the types used in a particular program. This section describes the content of that run-time data structure and provides operational specifications for the routines that maintain it. The discussion follows Section 6.1.2.5 of the standard.

9.1.1 Properties of types

The basic types are named in the OIL specification that describes the type model. A definition table key is used as the OIL name.

Every type has the `isComplete` property, whose value is `0` if the type is incomplete and `1` otherwise.

Properties of types[227] ≡

```
isComplete: int;
```

This macro is defined in definitions 227, 228, 230, 235, 237, 238, and 239.
This macro is invoked in definition 244.

9.1.2 Array type

If `T` is the element type of one or more array types, then those types have the key of an incomplete array type `T [ ]` as the value of their `ArrayOf` property. The array type `T [ ]` has the key of the element type `T` as the value of its `ElementType` property. This specification does not distinguish array types by size.

Properties of types[228] ≡

```
ElementType: DefTableKey;
ArrayOf:     DefTableKey;
```

This macro is defined in definitions 227, 228, 230, 235, 237, 238, and 239.
This macro is invoked in definition 244.

`KeyForArray` accepts the definition table key for an element type `T` and returns the corresponding incomplete array type:

Array type[229] ≡

```
if ((ArrayKey = GetArrayOf(T, NoKey)) == NoKey) {
  ArrayKey = NewKey();
  ResetOilType(
    ArrayKey,
    OilClassInst2(
      OilClassTypeIs_Array,
      ArrayKey,
      GetOilType(T, OilErrorType()),
      GetOilType(KeyForPointer(T), OilErrorType())));
  ResetArrayOf(T, ArrayKey);
  ResetElementType(ArrayKey, T);
}
```

This macro is invoked in definition 241.

9.1.3 Function type

If `T` is the type returned by one or more functions, then that type has the key of a function type `T ()` as the value of its `FunctionReturning` property. `T` is the value of the `ReturnType` property of the function type. This specification does not distinguish function types by argument signatures.

Properties of types[230] ≡

```
FunctionReturning: DefTableKey;
ReturnType       : DefTableKey;
```

This macro is defined in definitions 227, 228, 230, 235, 237, 238, and 239.
This macro is invoked in definition 244.

`KeyForFunction` accepts the definition table key for a return type `T` and returns the corresponding function type:

Function type[231] ≡

```
if ((FunctionKey = GetFunctionReturning(T, NoKey)) == NoKey) {
  tOilType functionType, pointerType;
```

```
  FunctionKey = NewKey();
  functionType=
    OilClassInst1(
      OilClassTypeIs_Function,
      FunctionKey,
      GetOilType(T, OilErrorType()));
  ResetOilType(FunctionKey,functionType);
  pointerType=GetOilType(KeyForPointer(FunctionKey), OilErrorType());
  OilAddCoercion(
    OilNewOp(
      CFunctoPtr,
      OilAddArgSig(pointerType, OilAddArgSig(functionType, OilNewArgSig())),
      1));
  ResetReturnType(FunctionKey, T);
  ResetFunctionReturning(T, FunctionKey);
  ResetKind(FunctionKey, Kind_function);
}
```

This macro is invoked in definition 241.

There are three different types of parameter lists. The first type is the `Unspecified` type, which corresponds to a non-ANSI declaration, where no signature is given. The second type, `ExplicitAndFixed` is the normal ANSI declaration, where the number of parameters given is explicit (i.e. it won't change). The third type is the variable declaration, and it's called `ExplicitAndVariable`.

Modifier types[232] ≡

```
typedef enum {
  Unspecified,
  ExplicitAndFixed,
  ExplicitAndVariable
} TypeOfSignature;
```

This macro is invoked in definition 240.

Attributes for type computations[233] ≡

```
ATTR  Type:        DefTableKey;
ATTR  FQSignature: int;
ATTR  Signature:   KeyArray;
```

This macro is invoked in definition 243.

`FQSignature` and `Signature` are used presently to determine the signature of a parameter list, but may change in the future if something more appropriate is found.

Parameter signature computations[234] ≡

```
SYMBOL parameter_part
COMPUTE
  SYNT.Signature   = NoKeyArray;
  SYNT.FQSignature = ExplicitAndFixed;
END;

SYMBOL parameter_type_list_opt
COMPUTE
```

```
  THIS.Signature   = NoKeyArray;
  THIS.FQSignature = ExplicitAndFixed;
END;
```

This macro is invoked in definition 243.

9.1.4 Pointer type

If `T` is the type referenced by a pointer, then that type has the key of the pointer type `T *` as the value of its `PointedToBy` property. `T` is the value of the `BaseType` property of the pointer type.

Properties of types[235] ≡

```
BaseType   : DefTableKey;
PointedToBy: DefTableKey;
```

This macro is defined in definitions 227, 228, 230, 235, 237, 238, and 239.
This macro is invoked in definition 244.

`KeyForPointer` accepts the definition table key for a referenced type `T` and returns the corresponding pointer type:

Pointer type[236] ≡

```
if ((PtrKey = GetPointedToBy(T, NoKey)) == NoKey) {
  PtrKey = NewKey();
  ResetOilType(
    PtrKey,
    OilClassInst1(
      OilClassTypeIs_Pointer,
      PtrKey,
      GetOilType(T, OilErrorType())));
  ResetisComplete(PtrKey, 1);
  ResetPointedToBy(T, PtrKey);
  ResetBaseType(PtrKey, T);
}
```

This macro is invoked in definition 241.

9.1.5 Qualified type

Any type so far mentioned is an *unqualified type*. Each unqualified type has three corresponding *qualified versions* of its type: a *const-qualified* version, a *volatile qualified* version, and a version having both qualifications. The qualified and unqualified versions of a type are distinct types that belong to the same type category.

If the unqualified version of a type has either the `isConst` or `isVolatile` property, their values are `0`. Qualified versions of the type have the appropriate properties with the value `1`. If a qualified version of a type has an inappropriate property, the value of that property is `0`. (Thus a const-qualified type has the `isConst` property with value `1`; it may or may not have the `isVolatile` property, but if that property is present then its value is `0`.)

Properties of types[237] ≡

```
isConst          : int;
isVolatile       : int;
```

This macro is defined in definitions 227, 228, 230, 235, 237, 238, and 239.

This macro is invoked in definition 244.

If qualified versions of a type exist, the unqualified version also exists. The definition table key for each qualified version is the value of the appropriate property (`ConstType`, `VolatileType` or `ConstVolatileType`) of the unqualified version. Every qualified version of a type has the definition table key of the unqualified version of that type as the value of the `UnqualifiedType` property.

Properties of types[238] ≡

```
UnqualifiedType   : DefTableKey;
ConstType         : DefTableKey;
VolatileType      : DefTableKey;
ConstVolatileType : DefTableKey;
```

This macro is defined in definitions 227, 228, 230, 235, 237, 238, and 239.
This macro is invoked in definition 244.

Properties of types[239] ≡

```
Type              : DefTableKey;
Kind              : int [Is];
```

This macro is defined in definitions 227, 228, 230, 235, 237, 238, and 239.
This macro is invoked in definition 244.

9.2 Specification files

9.2.1 buildtype.h

buildtype.h[240] ≡

```
#ifndef BUILDTYPE_H
#define BUILDTYPE_H

#include <stdio.h>
#include "eliproto.h"  /* determines if we can use prototypes */
#include "type.h"
#include "IntSet.h"
#include "pdl_gen.h"
#include "deftbl.h"
#include "err.h"     /* for error stuff */
#include "keyarray.h"
```

Modifier types[232]

```
extern DefTableKey KeyForArray         ELI_ARG((DefTableKey));
extern DefTableKey KeyForFunction      ELI_ARG((DefTableKey));
extern DefTableKey KeyForPointer       ELI_ARG((DefTableKey));
extern DefTableKey KeyForQualifier     ELI_ARG((DefTableKey, int, int));
extern DefTableKey KeyForEnum          ELI_ARG((DefTableKey));
extern DefTableKey KeyForStructOrUnion ELI_ARG((DefTableKey, TypeKinds));

#endif
```

This macro is attached to a product file.

9.2.2 buildtype.c

buildtype.c[241] ≡

```
#include "oiladt2.h"
#include "OilDecls.h"
#include "buildtype.h"

/*
 * This routine finds a qualified version of the current type
 *  The qualifier is specified by a Kwd_(const/volatile) parameter
 *
 * If there is an error, this routine calls the message()  function
 */

DefTableKey
#ifdef PROTO_OK
KeyForQualifier(DefTableKey currentKey, int isconst, int isvolatile)
#else
KeyForQualifier(currentKey, isconst, isvolatile)
DefTableKey currentKey; int isconst, isvolatile;
#endif
{ DefTableKey UnqualifiedKey, nextKey;

  UnqualifiedKey = GetUnqualifiedType(currentKey,currentKey);

  isconst    |= GetisConst(currentKey,0);
  isvolatile |= GetisVolatile(currentKey,0);

  if (isconst && isvolatile) {
    if ((nextKey = GetConstVolatileType(UnqualifiedKey,NoKey)) == NoKey) {
      ResetConstVolatileType(UnqualifiedKey, nextKey = NewKey());
      ResetOilType(nextKey, GetOilType(UnqualifiedKey, OilErrorType()));
      ResetisConst(nextKey, 1);
      ResetisVolatile(nextKey, 1);
      ResetUnqualifiedType(nextKey,UnqualifiedKey);
    }
  } else if (isconst) {
    if ((nextKey = GetConstType(UnqualifiedKey,NoKey)) == NoKey) {
      ResetConstType(UnqualifiedKey, nextKey = NewKey());
      ResetOilType(nextKey, GetOilType(UnqualifiedKey, OilErrorType()));
      ResetisConst(nextKey, 1);
      ResetUnqualifiedType(nextKey,UnqualifiedKey);
    }
  } else if (isvolatile) {
    if ((nextKey = GetVolatileType(UnqualifiedKey,NoKey)) == NoKey) {
      ResetVolatileType(UnqualifiedKey, nextKey = NewKey());
      ResetOilType(nextKey, GetOilType(UnqualifiedKey, OilErrorType()));
      ResetisVolatile(nextKey, 1);
      ResetUnqualifiedType(nextKey,UnqualifiedKey);
    }
  } else nextKey = UnqualifiedKey;   /* no qualifiers */
```

```
  return nextKey;
}

DefTableKey
#ifdef PROTO_OK
KeyForPointer(DefTableKey T)
#else
KeyForPointer(T) DefTableKey T;
#endif
{ DefTableKey  PtrKey;
```

Pointer type[236]

```
  return PtrKey;
}

DefTableKey
#ifdef PROTO_OK
KeyForArray(DefTableKey T)
#else
KeyForArray(T) DefTableKey T;
#endif
{ DefTableKey ArrayKey;
```

Array type[229]

```
  return ArrayKey;
}

DefTableKey
#ifdef PROTO_OK
KeyForFunction(DefTableKey T)
#else
KeyForFunction(T) DefTableKey T;
#endif
{ DefTableKey FunctionKey;
```

Function type[231]

```
  return FunctionKey;
}

DefTableKey
#ifdef PROTO_OK
KeyForStructOrUnion(DefTableKey TagKey, TypeKinds kind)
#else
KeyForStructOrUnion(TagKey, kind) DefTableKey TagKey; TypeKinds kind;
#endif
{ DefTableKey StructTypeKey;

  /* if a key for type is not present for this tag, create one */
  StructTypeKey = GetType(TagKey, NoKey);
  if (StructTypeKey == NoKey) {
```

```
    StructTypeKey = NewKey();

    /* reset the type property of the tag */
    ResetType(TagKey, StructTypeKey);

    /* reset the OIL type for the struct or union */
    ResetOilType(
      StructTypeKey,
      OilClassInst0(OilClassTypeIs_Struct,StructTypeKey));
  }

  IsKind(StructTypeKey, kind, Kind_error);
  return StructTypeKey;
}


/*
** This function returns a new DefTableKey for an enum.  All the
** corresponding properties are set for the returned key.
*/

/*
Section 6.5.3 states that all enumeration constants have the type int.
Thus, although every enum is a new type, all enums have an int OIL
type.
*/

DefTableKey
#ifdef PROTO_OK
KeyForEnum(DefTableKey TagKey)
#else
KeyForEnum(TagKey) DefTableKey TagKey;
#endif
{ DefTableKey EnumTypeKey;

  /* if a key for type is not present for this tag, create one */
  EnumTypeKey = GetType(TagKey, NoKey);
  if (EnumTypeKey == NoKey) {
    EnumTypeKey = NewKey();

    ResetType(TagKey, EnumTypeKey);
    ResetisComplete(EnumTypeKey, 1);

    /* add the new oil type for the enum */
    ResetOilType(EnumTypeKey, OilClassInst0(OilClassTypeIs_Enum, EnumTypeKey));
  }

  /* Check the kind of type, if there is no Kind property
     then the property is set to Kind_enum.  If it's not
     Kind_enum, change the type to Kind_error */
  IsKind(EnumTypeKey, Kind_enum, Kind_error);

  return EnumTypeKey;
```

```
}
```

This macro is attached to a product file.

9.2.3 buildtype.HEAD.phi

buildtype.HEAD.phi[242] ≡

```
#include "buildtype.h"
```

This macro is attached to a product file.

9.2.4 TypeRep.lido

TypeRep.lido[243] ≡

Attributes for type computations[233]
Parameter signature computations[234]

This macro is attached to a product file.

9.2.5 TypeRep.pdl

TypeRep.pdl[244] ≡

Properties of types[227]

This macro is attached to a product file.

9.2.6 TypeRep.specs

TypeRep.specs[245] ≡

```
$/pdl/keyarray.specs
```

This macro is attached to a product file.

10 Type Analysis

As discussed earlier, each type is represented by a definition table key. These keys are created in textual order, controlled by the chain `_C_GotTypes`. `_C_GotTypes` is an assertion that definition table keys are available for all types introduced to the left of the current textual position. (Some of these types may be incomplete, but their definition table keys have been assigned.) In addition, the void attribute `file.GotAllTypes` indicates that all type information has been extracted from the program's declarations:

Chain that indicates types have been set[246] ≡

```
CHAIN _C_GotTypes: VOID;

SYMBOL file COMPUTE
```

```
  CHAINSTART HEAD._C_GotTypes=0;
  SYNT.GotAllTypes=TAIL._C_GotTypes DEPENDS_ON THIS.MemberGotScopeProp;
END;

SYMBOL declarator COMPUTE
  THIS.Type=TAIL.TypeChain;
  THIS._C_GotTypes=THIS.Type;
END;

SYMBOL member_declarator COMPUTE
  THIS._C_GotTypes=TAIL.TypeChain;
END;

RULE: member_declarator ::=
COMPUTE
  member_declarator._C_GotTypes=member_declarator._C_GotTypes;
END;
```

This macro is invoked in definition 272.

10.1 Declarations

10.1.1 Storage-class specifiers

Declaration specifiers[247] ≡

Simple declaration specifier[262](‘`typedef`’)
Simple declaration specifier[262](‘`extern`’)
Simple declaration specifier[262](‘`static`’)
Simple declaration specifier[262](‘`auto`’)
Simple declaration specifier[262](‘`register`’)

This macro is defined in definitions 247, 250, 254, 260, 261, and 265.
This macro is invoked in definition 272.

Storage-class specifier keywords[248] ≡

```
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))
```

This macro is invoked in definition 256.

The `KWD` macro is used simply as a documentation aid, allowing us to pull together several aspects of the declaration specifiers. The first argument is the enumerated constant used to represent the declaration specifier internally, and the third gives the set of declaration specifiers that are incompatible with the specifier represented by the keyword. Abbreviations are used for specific groups of bits:

Abbreviations for sets of bits[249] ≡

```
#define ClassBits ((1<<(Kwd_register + 1)) - (1<<Kwd_typedef))
```

This macro is defined in definitions 249 and 251.

This macro is invoked in definition 269.

This ensures that at most one storage-class specifier may be given in the declaration specifiers of a declaration.

10.1.2 Type specifiers

Declaration specifiers[250] ≡

> *Specifier that determines type*[263](‘`void`’,‘`TypeIs_void`’)
> *Simple declaration specifier*[262](‘`char`’)
> *Simple declaration specifier*[262](‘`short`’)
> *Simple declaration specifier*[262](‘`int`’)
> *Simple declaration specifier*[262](‘`long`’)
> *Specifier that determines type*[263](‘`float`’,‘`TypeIs_float`’)
> *Simple declaration specifier*[262](‘`double`’)
> *Simple declaration specifier*[262](‘`signed`’)
> *Simple declaration specifier*[262](‘`unsigned`’)

This macro is defined in definitions 247, 250, 254, 260, 261, and 265.
This macro is invoked in definition 272.

As in the case of storage specifiers, the `KWD` macro is used to describe the keywords and the constraints on their use. The abbreviations are:

Abbreviations for sets of bits[251] ≡

```
#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 defined in definitions 249 and 251.
This macro is invoked in definition 269.

Type specifier keywords[252] ≡

```
/*      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)),
```

This macro is defined in definitions 252 and 253.
This macro is invoked in definition 256.

A `struct_or_union_specifier`, `enum_specifier` or `typedef_name` is represented by the keyword `Kwd_typeid`. Like `void` and `float`, these specifiers completely determine the type and may not occur with any other type specifiers:

Type specifier keywords[253] ≡

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

This macro is defined in definitions 252 and 253.

This macro is invoked in definition 256.

In general, all of the specifiers must be examined before the specified type is known. We use a chain, `Specification`, to implement the state of the specifier scan:

Declaration specifiers[254] ≡

```
CHAIN Specification: SpecData;
```

This macro is defined in definitions 247, 250, 254, 260, 261, and 265.
This macro is invoked in definition 272.

Specification data[255] ≡

```
typedef struct {
  long KeywordSet;
  DefTableKey SpecifiedType;
  int CurrentState;
} SpecData;
```

This macro is invoked in definition 274.

Specifier keywords[256] ≡

Type specifier keywords[252],
Storage-class specifier keywords[248],

```
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 259 and 269.

The `KWD` macro is used simply as a documentation aid, allowing us to pull together several aspects of the declaration 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:

The finite-state machine is defined by the following transition table:

Finite-state machine[257] ≡

```
/*                         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, 0, 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}
```

This macro is invoked in definition 277.

State 0 is the initial state, and if the state of the machine determines the type, the type determined is specified by the second component of the state.

The finite-state machine is implemented as part of an abstract data type that exports the operations `InitSpecifiers`, `FinalType` and `InSpecifierSet` to its customers.

Abstract data type for finite-state machine[258] ≡

```
SpecData
#ifdef PROTO_OK
InitSpecifiers(void)
#else
InitSpecifiers()
#endif
{ SpecData CurrentData;
  CurrentData.KeywordSet    = 0;
  CurrentData.SpecifiedType = NoKey;
  CurrentData.CurrentState  = 0;
  return CurrentData; }

DefTableKey
#ifdef PROTO_OK
FinalType(SpecData chain)
#else
FinalType(chain) SpecData chain;
#endif
{ if (chain.SpecifiedType != NoKey) return chain.SpecifiedType;
  else return State[chain.CurrentState].Type;
}
```

This macro is defined in definitions 258 and 259.
This macro is invoked in definition 277.

Another operation, `NextSpecifier`, is exported by the module. This operation takes the keyword for the specifier and the chain value. It returns `1` if the specifier is legal in the current state of the machine, and `0` otherwise. It also caches the updated chain value for subsequent operations.

Abstract data type for finite-state machine[259] ≡

```
#define KWD(w,i,s) i
static int FSMInput[] = {
Specifier keywords[256]
};
#undef KWD

#define KWD(w,i,s) s
static long Exclude[] = {
```

```
Specifier keywords[256]
};
#undef KWD

int
#ifdef PROTO_OK
NextSpecifier(TypeSpecifier kw, SpecData chain)
#else
NextSpecifier(kw, chain) TypeSpecifier kw; SpecData chain;
#endif
{ return (Exclude[kw] & chain.KeywordSet) == 0; }

SpecData
#ifdef PROTO_OK
UpdateSpecification(DefTableKey type, TypeSpecifier kw, SpecData chain)
#else
UpdateSpecification(type, kw, chain)
DefTableKey type; TypeSpecifier kw; SpecData chain;
#endif
{ 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 defined in definitions 258 and 259.
This macro is invoked in definition 277.

Declaration specifiers[260] ≡

```
SYMBOL declaration_specifiers COMPUTE
  CHAINSTART HEAD.Specification=InitSpecifiers() DEPENDS_ON THIS._C_GotTypes;
  SYNT.TypeSpecified=FinalType(TAIL.Specification);
  SYNT.IsTypedef= InSpecifierSet(Kwd_typedef, TAIL.Specification);
  SYNT.IsExtern=  InSpecifierSet(Kwd_extern,  TAIL.Specification);
  SYNT.IsStatic=  InSpecifierSet(Kwd_static,  TAIL.Specification);
  SYNT.IsRegister=InSpecifierSet(Kwd_register,TAIL.Specification);
  SYNT.IsConst=   InSpecifierSet(Kwd_const,   TAIL.Specification);
  SYNT.IsVolatile=InSpecifierSet(Kwd_volatile,TAIL.Specification);
  THIS._C_GotTypes="yes"
    DEPENDS_ON (
      THIS.TypeSpecified,
      THIS.IsTypedef,
      THIS.IsExtern,
      THIS.IsStatic,
      THIS.IsRegister,
      THIS.IsConst,
      THIS.IsVolatile);
END;
```

This macro is defined in definitions 247, 250, 254, 260, 261, and 265.
This macro is invoked in definition 272.

NextSpecifier is used to set the attribute ok which, in turn, controls the production of an error report:

Declaration specifiers[261] ≡

```
ATTR ok: int;

SYMBOL declaration_specifier COMPUTE
  IF(NOT(THIS.ok),
    message(ERROR,"Illegal specifier",0,COORDREF));
END;
```

This macro is defined in definitions 247, 250, 254, 260, 261, and 265.
This macro is invoked in definition 272.

The computation of ok is the only task to be done for most of the declaration specifiers, but these computations must be synchronized via Specification:

Simple declaration specifier[262](◇1) ≡

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

This macro is invoked in definitions 247, 250, and 265.

Here the parameter is the C keyword.

Five type specifiers determine the type immediately, and cannot appear with size or sign specifiers.

Specifier that determines type[263](◇2) ≡

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

This macro is invoked in definition 250.

type for ds1 with a typedef_name[264] ≡

```
RULE: declaration_specifier ::= typedef_name
COMPUTE
  declaration_specifier.ok=
    NextSpecifier(Kwd_typeid,declaration_specifier.Specification);
  declaration_specifier.Specification=
    UpdateSpecification(
      GetType(OrdinaryIdStackArray(typedef_name), NoKey),
      Kwd_typeid,
      declaration_specifier.Specification);
END;
```

This macro is invoked in definition 272.

10.1.3 Type Qualifiers

Declaration specifiers[265] ≡

> *Simple declaration specifier*[262](‘`const`’)
> *Simple declaration specifier*[262](‘`volatile`’)

This macro is defined in definitions 247, 250, 254, 260, 261, and 265.
This macro is invoked in definition 272.

Since incompleteness of a type depends on processing the types in textual order, we need to ensure this order is maintained during the evaluation of the types by using the `_C_GotTypes` chain.

If a `struct_or_union_specifier` contains a member specification, the computations create a new type in accordance to the standard section 6.5.2.3. The new type is a struct or union containing the members of `component_list`. The list of members is created by consistent renaming over member scopes and is available in `component_list.MemberEnv`.

type for ds1 with a struct_or_union_specifier[266] ≡

```
RULE: declaration_specifier ::= struct_or_union_specifier
COMPUTE
  declaration_specifier.ok=
    NextSpecifier(Kwd_typeid,struct_or_union_specifier.Specification);
  declaration_specifier.Specification=
    UpdateSpecification(
      struct_or_union_specifier.Type,
      Kwd_typeid,
      struct_or_union_specifier.Specification);
END;

ATTR IsStruct: TypeKinds;

RULE: struct_or_union ::= 'struct'
COMPUTE
  struct_or_union.IsStruct=Kind_struct;
END;

RULE: struct_or_union ::= 'union'
COMPUTE
  struct_or_union.IsStruct=Kind_union;
END;
```

This macro is invoked in definition 272.

type for ds1 with a enum_specifier[267] ≡

```
RULE: declaration_specifier ::= enum_specifier
COMPUTE
  declaration_specifier.ok=
    NextSpecifier(Kwd_typeid,declaration_specifier.Specification);
  declaration_specifier.Specification=
    UpdateSpecification(
      enum_specifier.Type,
      Kwd_typeid,
      declaration_specifier.Specification)
```

```
    DEPENDS_ON enum_specifier.Specification;
END;
```

This macro is invoked in definition 272.

Type kinds[268] ≡

```
typedef enum {
  Kind_error,
  Kind_struct,
  Kind_union,
  Kind_enum,
  Kind_function,
} TypeKinds;
```

This macro is invoked in definition 274.

Type keywords[269] ≡

```
#define KWD(w,i,s) w
typedef enum {
Specifier keywords[256]
} TypeSpecifier;
#undef KWD

Abbreviations for sets of bits[249]
```

This macro is invoked in definition 274.

typesets code[270] ≡

```
TypeIs_void             -> PointedToBy={TypeIs_VoidPointer};
TypeIs_VoidPointer      -> BaseType={TypeIs_void};

/* void is incomplete according to the standard */
TypeIs_char             -> isComplete={1};
TypeIs_signed_char      -> isComplete={1};
TypeIs_unsigned_char    -> isComplete={1};
TypeIs_short            -> isComplete={1};
TypeIs_unsigned_short   -> isComplete={1};
TypeIs_int              -> isComplete={1};
TypeIs_unsigned_int     -> isComplete={1};
TypeIs_long             -> isComplete={1};
TypeIs_unsigned_long    -> isComplete={1};
TypeIs_float            -> isComplete={1};
TypeIs_double           -> isComplete={1};
TypeIs_long_double      -> isComplete={1};
```

This macro is invoked in definition 273.

Set library[271] ≡

```
$/Adt/IntSet.gnrc :inst
$/Adt/List.gnrc+instance=DefTableKey +referto=deftbl :inst
```

This macro is invoked in definition 275.

10.2 Specification Files

10.2.1 type.lido

type.lido[272] ≡

> *Chain that indicates types have been set*[246]
> *Declaration specifiers*[247]
> *type for ds1 with a typedef_name*[264]
> *type for ds1 with a struct_or_union_specifier*[266]
> *type for ds1 with a enum_specifier*[267]

This macro is attached to a product file.

10.2.2 type.pdl

type.pdl[273] ≡

> *typesets code*[270]

This macro is attached to a product file.

10.2.3 type.h

type.h[274] ≡

```
#ifndef TYPE_H
#define TYPE_H

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

#define UNSIZEDARRAY (0)

struct ArrayTuple
{
        DefTableKey TheKey;
        int     Size;
        struct ArrayTuple *Next;
};

typedef struct ArrayTuple *ArrayList;
#define NoArrayList ((ArrayList) 0)

extern void AddToArrayList ELI_ARG((ArrayList *, DefTableKey, int));
extern DefTableKey FindArraySized ELI_ARG((ArrayList, int));
```

> *Type kinds*[268]
> *Type keywords*[269]

Specification data[255]

```
extern SpecData InitSpecifiers ELI_ARG((void));
extern SpecData
  UpdateSpecification ELI_ARG((DefTableKey, TypeSpecifier, SpecData));
extern DefTableKey FinalType ELI_ARG((SpecData));
extern int NextSpecifier ELI_ARG((TypeSpecifier, SpecData));

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

#endif
```

This macro is attached to a product file.

10.2.4 type.specs

type.specs[275] ≡

Set library[271]

```
$/oil/oiladt2.h
```

This macro is attached to a product file.

10.2.5 type.head

type.head[276] ≡

```
#include "type.h"
```

This macro is attached to a product file.

10.2.6 type.c

type.c[277] ≡

```
#include "type.h"
#include "oiladt2.h"
#include "OilDecls.h"
#include "buildtype.h"

static struct {int Next[7]; DefTableKey Type;} State[] = {
```

Finite-state machine[257]

```
};
```

Abstract data type for finite-state machine[258]

This macro is attached to a product file.

11 Tree Type Computations

This section is for all tree computations of expressions.

11.1 Type computations in expressions

This section of the document does the type computations for expressions, to determine the needed types, type coercions, constant value calculations, etc.

11.1.1 Calculate possible and final types

The following computations do type calculations for expressions, to determine the operators that are needed. Note that According to section 6.4 of the standard, "Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within the operand of a sizeof operator." This is why comma expressions are never treated as constant expressions.

Rule computations[278] ≡

```
ATTR op           : tOilOp;       /* attributes only used locally */

ATTR PossTypes    : tOilTypeSet;  /* possible types.  A synthesized attribute */
ATTR FinalType    : tOilType;     /* final type.  an inherited attribute      */
ATTR Indication   : tOilOp;

ATTR ExprValue    : int;

SYMBOL Expression:
  IsKnownConst : int,
  NeedaFunc    : int;          /* true for function calls */

SYMBOL file COMPUTE
  SYNT.GotAllConstantId=CONSTITUENTS enumeration_constant.GotConstantId;
END;

SYMBOL enumeration_constant COMPUTE
  SYNT.GotConstantId=ResetIsKnownConst(THIS.Key, 1);
END;

SYMBOL Expression COMPUTE
  SYNT.IsKnownConst=0;
  SYNT.ExprValue=0;
  INH.NeedaFunc=0;
  INH.FinalType=OilSelectTypeFromTS(THIS.PossTypes);
END;

RULE: constant_exp_opt ::=
COMPUTE
  constant_exp_opt.ExprValue = 0;
END;

RULE: Expression ::= IdUse
COMPUTE
  Expression.PossTypes=
    IF (EQ(Expression.NeedaFunc, 1),
      GetTypeWhenIdUseIsaFunction(IdUse.Key, COORDREF),
      GetTypeWhenIdUseIsDefined(IdUse.Key, COORDREF))
```

```
    DEPENDS_ON INCLUDING file.GotAllTypes;
  Expression.IsKnownConst=
    GetIsKnownConst(IdUse.Key, 0)
    DEPENDS_ON INCLUDING file.GotAllConstantId;
END;

RULE: Expression ::= constant
COMPUTE
  Expression.PossTypes = constant.PossTypes;
  Expression.ExprValue = constant.ExprValue;
  Expression.IsKnownConst = 1;
END;

RULE: Expression ::= StringSeq
COMPUTE
  Expression.PossTypes=
    OilTypeToSet(
      GetOilType(
        KeyForQualifier(KeyForPointer(TypeIs_char), 1, 0),
        OilErrorType()));
  Expression.IsKnownConst = 1;
END;

RULE: Expression ::= Expression '[' Expression ']'
COMPUTE
  .op=
    OilIdOpTS2(
      Expression[1].FinalType,
      OilOpSubscript_Indication,
      Expression[2].PossTypes,
      Expression[3].PossTypes);
  Expression[1].PossTypes=
    OilIdResultTS2(
      OilOpSubscript_Indication,
      Expression[2].PossTypes,
      Expression[3].PossTypes);
  Expression[1].IsKnownConst=
    AND(Expression[2].IsKnownConst,Expression[3].IsKnownConst);
  Expression[2].FinalType=OilGetArgType(.op,1);
  Expression[3].FinalType=OilGetArgType(.op,2);
END;

RULE: Expression ::= Expression '(' argument_exp_list ')'
COMPUTE
  .op=
    OilIdOpTS1(
      Expression[1].FinalType,
      OilOpCall_Indication,
      Expression[2].PossTypes);
IF(NOT(OilIsValidOp(.op)),message(ERROR,"Invalid call",0,COORDREF));
  /* check arg list types */
  Expression[1].PossTypes=
    OilIdResultTS1(OilOpCall_Indication, Expression[2].PossTypes);
```

```
  Expression[2].NeedaFunc=1;
END;

RULE: Expression ::= Expression '(' ')'
COMPUTE
  /* check arg list types */
  Expression[1].PossTypes=
    OilIdResultTS1(OilOpCall_Indication, Expression[2].PossTypes);
  Expression[2].NeedaFunc=1;
END;

ATTR type: DefTableKey;
ATTR kind: TypeKinds;

RULE: Expression ::= Expression '.' MemberIdUse
COMPUTE
  .type=OilTypeName(OilSelectTypeFromTS(Expression[2].PossTypes));
  .kind=GetKind(.type,Kind_error);
  Expression[1].PossTypes =
    OilTypeToSet(
      GetOilType(
        GetType(MemberIdUse.MemberKey,NoKey),
        OilErrorType()));
  MemberIdUse.MemberScope=GetMemberScope(.type,NoEnv);
  IF(AND(NE(.kind,Kind_struct),NE(.kind,Kind_union)),
    message(ERROR,"Not a structure or union",0,COORDREF));
END;

RULE: Expression ::= Expression '->' MemberIdUse
COMPUTE
  .type=
    GetBaseType(
      OilTypeName(OilSelectTypeFromTS(Expression[2].PossTypes)),
      NoKey);
  .kind=GetKind(.type,Kind_error);
  Expression[1].PossTypes=
    OilTypeToSet(
      GetOilType(
        GetType(MemberIdUse.MemberKey,NoKey),
        OilErrorType()));
  MemberIdUse.MemberScope=GetMemberScope(.type,NoEnv);
  IF(AND(NE(.kind,Kind_struct),NE(.kind,Kind_union)),
    message(ERROR,"Not a pointer to a structure or union",0,COORDREF));
END;

RULE: Expression ::= Expression '++'
COMPUTE
  .op=
    OilIdOpTS1(
      Expression[1].FinalType,
      OilOpIncrement_Indication,
      Expression[2].PossTypes);
  Expression[1].PossTypes=Expression[2].PossTypes;
```

```
  Expression[2].FinalType=OilGetArgType(.op,1);
END;

RULE: Expression ::= Expression '--'
COMPUTE
  .op=
    OilIdOpTS1(
      Expression[1].FinalType,
      OilOpDecrement_Indication,
      Expression[2].PossTypes);
  Expression[1].PossTypes=Expression[2].PossTypes;
  Expression[2].FinalType=OilGetArgType(.op, 1);
END;

RULE: Expression ::= '++' Expression
COMPUTE
  .op=
    OilIdOpTS1(
      Expression[1].FinalType,
      OilOpIncrement_Indication,
      Expression[2].PossTypes);
  Expression[1].PossTypes=Expression[2].PossTypes;
  Expression[2].FinalType=OilGetArgType(.op,1);
END;

RULE: Expression ::= '--' Expression
COMPUTE
  .op=
    OilIdOpTS1(
      Expression[1].FinalType,
      OilOpDecrement_Indication,
      Expression[2].PossTypes);
  Expression[1].PossTypes=Expression[2].PossTypes;
  Expression[2].FinalType=OilGetArgType(.op,1);
END;

RULE: Expression ::= unary_operator Expression
COMPUTE
  .op=unary_operator.op;
  Expression[1].PossTypes=
    OilIdResultTS1(unary_operator.Indication,Expression[2].PossTypes);
  unary_operator.op=
    OilIdOpTS1(
      Expression[1].FinalType,
      unary_operator.Indication,
      Expression[2].PossTypes);
  Expression[1].IsKnownConst=
    IF(
      OR(
        EQ(unary_operator.Indication,OilOpDereference_Indication),
        EQ(unary_operator.Indication,OilOpReference_Indication)),
      0,
      Expression[2].IsKnownConst);
```

```
  Expression[2].FinalType=OilGetArgType(unary_operator.op,1);
END;

RULE: Expression ::= 'sizeof' Expression
COMPUTE
  Expression[1].PossTypes=OilTypeToSet(OilTypeTypeIs_int);
  Expression[1].IsKnownConst=1;
END;

RULE: Expression ::= 'sizeof' '(' type_name ')'
COMPUTE
  Expression[1].PossTypes=OilTypeToSet(OilTypeTypeIs_int);
  Expression[1].IsKnownConst=1;
END;

RULE: Expression ::= '(' type_name ')' Expression
COMPUTE
  .op=
    OilIdOpTS1(
      GetOilType(type_name.Type,OilErrorType()),
      OilOpCast_Indication,
      Expression[2].PossTypes);
  Expression[1].PossTypes=
    OilTypeToSet(GetOilType(type_name.Type,OilErrorType()));
  Expression[1].IsKnownConst=Expression[2].IsKnownConst;
  Expression[1].ExprValue=Expression[2].ExprValue;
  Expression[2].FinalType=OilGetArgType(.op,1);
END;

RULE: Expression ::= Expression binary_operator Expression
COMPUTE
  Expression[1].PossTypes=
    OilIdResultTS2(
      binary_operator.Indication,
      Expression[2].PossTypes,
      Expression[3].PossTypes);
  Expression[1].IsKnownConst=
    AND(Expression[2].IsKnownConst,Expression[3].IsKnownConst);
  binary_operator.op=
    OilIdOpTS2(
      Expression[1].FinalType,
      binary_operator.Indication,
      Expression[2].PossTypes,
      Expression[3].PossTypes);
  Expression[2].FinalType=OilGetArgType(binary_operator.op,1);
  Expression[3].FinalType=OilGetArgType(binary_operator.op,2);
END;

RULE: Expression ::= Expression '?' Expression ':' Expression
COMPUTE
  .op=
    OilIdOpTS1(
      OilTypeTypeIs_int,
```

```
        OilOpConditional_Indication,
        Expression[2].PossTypes);
    Expression[1].PossTypes=
      OilTypeToSet(OilBalance(Expression[3].PossTypes,Expression[4].PossTypes));
    Expression[1].IsKnownConst=
      AND(
        Expression[2].IsKnownConst,
        AND(Expression[3].IsKnownConst,Expression[4].IsKnownConst));
    Expression[2].FinalType=OilGetArgType(.op,1);
    Expression[3].FinalType=Expression[1].FinalType;
    Expression[4].FinalType=Expression[1].FinalType;
  END;

  RULE: Expression ::= Expression ',' Expression
  COMPUTE
    Expression[1].PossTypes=Expression[3].PossTypes;
    Expression[3].FinalType=Expression[1].FinalType;
  END;
```

This macro is invoked in definition 293.

Miscellaneous Types[279] ≡

```
  ATTR FloatValue: int;
  RULE: constant ::=  floating_constant
  COMPUTE
    constant.PossTypes=OilTypeToSet(OilTypeTypeIs_float);
    constant.ExprValue=floating_constant;
  END;

  ATTR IntegerValue: int;
  RULE: constant ::= integer_constant
  COMPUTE
    constant.ExprValue=integer_constant;
    constant.PossTypes=
      IF(EQ(integer_constant,0),
        OilTypeToSet(OilTypeTypeIs_NULL),
        OilTypeToSet(OilTypeTypeIs_int));
  END;

  ATTR CharValue: int;
  RULE: constant ::=  character_constant
  COMPUTE
    constant.PossTypes=OilTypeToSet(OilTypeTypeIs_char);
    constant.ExprValue=character_constant;
  END;
```

This macro is invoked in definition 293.

This section is the "shut up the compiler" section.

Shut Up[280] ≡

```
  RULE: struct_declarator ::=  member_declarator ':' Expression
  COMPUTE
```

```
  /* specify that Expression is an integral constant */
  Expression.FinalType = OilTypeTypeIs_int;
END;

RULE: enumerator ::=  enumeration_constant '=' Expression
COMPUTE
  /* specify that Expression is an integral constant */
  Expression.FinalType = OilTypeTypeIs_unsigned_long;
END;

RULE: labeled_statement ::=  'case' Expression ':' statement
COMPUTE
  /* specify that Expression is an integral constant */
  Expression.FinalType = OilTypeTypeIs_int;
END;

RULE: Expression ::=
COMPUTE
  Expression.PossTypes=OilTypeToSet(OilTypeTypeIs_int);
END;
```

This macro is invoked in definition 293.

Required types[281] ≡

```
RULE: selection_statement ::= 'if' '(' Expression ')' statement
COMPUTE
  .op=
    OilIdOpTS1(
      OilTypeTypeIs_int,
      OilOpConditional_Indication,
      Expression.PossTypes);
END;

RULE: selection_statement ::= 'if' '(' Expression ')' statement
                               'else' statement
COMPUTE
  .op=
    OilIdOpTS1(
      OilTypeTypeIs_int,
      OilOpConditional_Indication,
      Expression.PossTypes);
END;

RULE: selection_statement ::= 'switch' '(' Expression ')' statement
COMPUTE
  .op=
    OilIdOpTS1(
      OilTypeTypeIs_int,
      OilOpConditional_Indication,
      Expression.PossTypes);
END;

RULE: iteration_statement ::= 'while' '(' Expression ')' statement
```

```
COMPUTE
  .op=
    OilIdOpTS1(
      OilTypeTypeIs_int,
      OilOpConditional_Indication,
      Expression.PossTypes);
END;

RULE: iteration_statement ::= 'do' statement 'while' '(' Expression ')' ';'
COMPUTE
  .op=
    OilIdOpTS1(
      OilTypeTypeIs_int,
      OilOpConditional_Indication,
      Expression.PossTypes);
END;

RULE: jump_statement ::= 'return' Expression ';'
COMPUTE
  .op=
    OilIdOp2(
      OilOpAssign_Indication,
      GetOilType(
        GetReturnType(INCLUDING function_definition.Type, NoKey),
        OilErrorType()),
      OilSelectTypeFromTS(Expression.PossTypes));
  Expression.FinalType=OilGetArgType(.op, 2);
END;

RULE: for_test ::=  Expression
COMPUTE
  .op=
    OilIdOpTS1(
      OilTypeTypeIs_int,
      OilOpConditional_Indication,
      Expression.PossTypes);
END;
```

This macro is invoked in definition 293.

Get return type of function_definition[282] ≡

```
RULE: function_definition ::=
     declaration_specifiers declarator declaration_list_opt compound_statement
COMPUTE
  function_definition.Type = declarator.TypeChain;
END;

RULE: function_definition ::=
     declarator declaration_list_opt compound_statement
COMPUTE
  function_definition.Type = declarator.TypeChain;
END;
```

This macro is invoked in definition 293.

11.1.2 Calculate Operator Indications

The following macro can be used to specify all of the rules for the indications.

Assign Indication[283](◇3) ≡

```
RULE: ◇1 ::=  ◇2
COMPUTE
  ◇1.Indication = OilOp◇3_Indication;
END;
```

This macro is invoked in definitions 284 and 285.

Unary Indication rules[284] ≡

Assign Indication[283](‘unary_operator’,‘’&’’,‘Reference’)
Assign Indication[283](‘unary_operator’,‘’*’’,‘Dereference’)
Assign Indication[283](‘unary_operator’,‘’+’’,‘Plus’)
Assign Indication[283](‘unary_operator’,‘’-’’,‘Minus’)
Assign Indication[283](‘unary_operator’,‘’~’’,‘Bitwise_Not’)
Assign Indication[283](‘unary_operator’,‘’!’’,‘Not’)

This macro is invoked in definition 293.

Binary Indication rules[285] ≡

Assign Indication[283](‘binary_operator’,‘’*’’,‘Multiplication’)
Assign Indication[283](‘binary_operator’,‘’/’’,‘Division’)
Assign Indication[283](‘binary_operator’,‘’%’’,‘Mod’)
Assign Indication[283](‘binary_operator’,‘’+’’,‘Addition’)
Assign Indication[283](‘binary_operator’,‘’-’’,‘Subtraction’)
Assign Indication[283](‘binary_operator’,‘’<<’’,‘Bit_Shift_Left’)
Assign Indication[283](‘binary_operator’,‘’>>’’,‘Bit_Shift_Right’)
Assign Indication[283](‘binary_operator’,‘’<’’,‘LessThan’)
Assign Indication[283](‘binary_operator’,‘’>’’,‘Greater’)
Assign Indication[283](‘binary_operator’,‘’<=’’,‘LessThan_Equal’)
Assign Indication[283](‘binary_operator’,‘’>=’’,‘Greater_Equal’)
Assign Indication[283](‘binary_operator’,‘’==’’,‘Equality’)
Assign Indication[283](‘binary_operator’,‘’!=’’,‘Not_Equal’)
Assign Indication[283](‘binary_operator’,‘’&’’,‘Bitwise_And’)
Assign Indication[283](‘binary_operator’,‘’^’’,‘Bitwise_XOr’)
Assign Indication[283](‘binary_operator’,‘’|’’,‘Bitwise_Or’)
Assign Indication[283](‘binary_operator’,‘’&&’’,‘And’)
Assign Indication[283](‘binary_operator’,‘’||’’,‘Or’)
Assign Indication[283](‘binary_operator’,‘’=’’,‘Assign’)
Assign Indication[283](‘binary_operator’,‘’*=’’,‘Mult_Eq’)
Assign Indication[283](‘binary_operator’,‘’/=’’,‘Div_Eq’)
Assign Indication[283](‘binary_operator’,‘’%=’’,‘Mod_Eq’)
Assign Indication[283](‘binary_operator’,‘’+=’’,‘Plus_Eq’)
Assign Indication[283](‘binary_operator’,‘’-=’’,‘Minus_Eq’)
Assign Indication[283](‘binary_operator’,‘’<<=’’,‘Bitwise_Shift_Left_Eq’)
Assign Indication[283](‘binary_operator’,‘’>>=’’,‘Bitwise_Shift_Right_Eq’)
Assign Indication[283](‘binary_operator’,‘’&=’’,‘Bitwise_And_Eq’)
Assign Indication[283](‘binary_operator’,‘’^=’’,‘Bitwise_XOr_Eq’)

Assign Indication[283](‘binary_operator’,‘’|=’’,‘Bitwise_Or_Eq’)

This macro is invoked in definition 293.

11.2 Error Checking

The following rules do sanity checks on the above computations, to check for error conditions.

Sanity Checks for constant values[286] ≡

```
RULE: constant_exp_opt ::= Expression
COMPUTE
  constant_exp_opt.ExprValue = Expression.ExprValue;
  IF(NOT(Expression.IsKnownConst),
    message(ERROR,"Array bound is not constant",0,COORDREF));
END;

RULE: struct_declarator ::= member_declarator ':' Expression
COMPUTE
  IF(NOT(Expression.IsKnownConst),
    message(ERROR,"Bitfield size is not constant",0,COORDREF));
END;

RULE: enumerator ::= enumeration_constant '=' Expression
COMPUTE
  IF(NOT(Expression.IsKnownConst),
    message(ERROR,"Enumeration value is not constant",0,COORDREF));
END;

RULE: labeled_statement ::= 'case' Expression ':' statement
COMPUTE
  IF(NOT(Expression.IsKnownConst),
    message(ERROR,"case value is not constant",0,COORDREF));
END;
```

This macro is invoked in definition 293.

This operator should eliminate some typing, because most of the checks are very repetitive.

Operator Check[287](◇2) ≡

```
RULE: ◇1
COMPUTE
  IF (EQ(.op, OilErrorOp()),
        message(ERROR, ◇2, 0, COORDREF));
END;
```

This macro is invoked in definition 288.

Sanity checks for expression types[288] ≡

Operator Check[287](‘Expression ::= Expression '[' Expression ']'’,‘
"Not an array type, or invalid subscript"’)

Operator Check[287](‘Expression ::= Expression '++' ’,‘
"Cannot use increment operator on non-scalar types"’)

Operator Check[287](‘Expression ::= Expression '--' ’,‘
"Cannot use decrement operator on non-scalar types"’)

Operator Check[287](‘Expression ::= '++' Expression ’,‘
"Cannot use increment operator on non-scalar types"’)

Operator Check[287](‘Expression ::= '--' Expression ’,‘
"Cannot use decrement operator on non-scalar types"’)

```
SYMBOL operator COMPUTE
  IF(NOT(OilIsValidOp(THIS.op)),
    message(ERROR, "Invalid operator", 0, COORDREF));
END;

SYMBOL unary_operator INHERITS operator END;
SYMBOL binary_operator INHERITS operator END;
```

Operator Check[287](‘Expression ::= '(' type_name ')' Expression ’,‘
"Illegal cast"’)

Operator Check[287](‘Expression ::= Expression '?' Expression ':' Expression ’,‘
"conditional expression not a valid type"’)

Operator Check[287](‘selection_statement ::= 'if' '(' Expression ')' statement ’,‘
"Illegal expression in if condition"’)

Operator Check[287](‘selection_statement ::= 'if' '(' Expression ')'
statement 'else' statement ’,‘
"Illegal expression in if condition"’)

Operator Check[287](‘selection_statement ::= 'switch' '(' Expression ')'
statement ’,‘ "Illegal expression in switch condition"’)

Operator Check[287](‘iteration_statement ::= 'while' '(' Expression ')'
statement ’,‘ "Illegal expression in while condition"’)

Operator Check[287](‘iteration_statement ::= 'do' statement 'while'
'(' Expression ')' ';' ’,‘
"Illegal expression in while condition"’)

Operator Check[287](‘for_test ::= Expression ’,‘
"Illegal expression in while condition"’)

```
RULE: Expression ::= Expression '?' Expression ':' Expression
COMPUTE
  IF (EQ(OilSelectTypeFromTS(Expression[1].PossTypes), OilErrorType()),
        message(ERROR,
         "The expressions for ':' have incompatible types",
         0, COORDREF));
```

```
END;
```

This macro is invoked in definition 293.

11.2.1 Code to handle previously undefined variables

The following code is used to do two things: make sure that "normal" uses of variables are declared before use, and to make sure that when call are made to funtions that were perviously undeclared, the `DefTableKey` for that variable is properly set up. This means guaranteeing that the `DefTableKey` has an `OilType`, etc., etc...

Guarantee DefTableKey for function[289] ≡

```
tOilTypeSet
#ifdef PROTO_OK
GetTypeWhenIdUseIsaFunction(DefTableKey functionKey, CoordPtr curpos)
#else
GetTypeWhenIdUseIsaFunction(functionKey, curpos)
  DefTableKey functionKey;
  CoordPtr curpos;
#endif
{
  /*
   *  Comment: currently, the only context this is being called from is
   *    when we have been handed a DefTableKey (of an identifier), and we
   *    know that this DefTableKey has to have a function type.
   */

  if (GetType(functionKey, NoKey) == NoKey) {

    /*
     *  If we get to here, this key doesn't have a Type property set.
     *    I take this to mean that this is the 1st time we've seen it,
     *    which means that it gets turned into an incomplete prototype
     */
    ResetType(functionKey, KeyForFunction(TypeIs_int));
    ResetKind(functionKey, Kind_function);
  }

  /*
   *  Make sure that we're working with a function
   */
  if (GetKind( GetType(functionKey, NoKey), Kind_error ) != Kind_function) {
    message(ERROR, "Illegal function", 0, curpos);
    return OilTypeToSet(OilErrorType());
  }

  return OilTypeToSet(GetOilType(GetType(functionKey,NoKey),OilErrorType()));
}
```

This macro is invoked in definition 292.

This code makes sure that the given `DefTableKey` has been declared, and that this is not the 1st occurrence of it.

Guarantee declaration for DefTableKey[290] ≡

```
tOilTypeSet
#ifdef PROTO_OK
GetTypeWhenIdUseIsDefined (DefTableKey variableKey, CoordPtr curpos)
#else
GetTypeWhenIdUseIsDefined (variableKey, curpos)
  DefTableKey variableKey;
  CoordPtr curpos;
#endif
{ TypeKinds kind;
  DefTableKey IdType;

  if (GetType(variableKey, NoKey) == NoKey) {
    /*
     * This may be wrong.... but it prevents some cascade errors if we
     *   use this variable in other places
     */
    ResetType(variableKey, TypeIs_int);
    ResetReturnType(variableKey, TypeIs_int);
  }

  IdType = GetType(variableKey, NoKey);
  return
    OilTypeToSet(
      GetOilType(
        GetKind(IdType, Kind_error) == Kind_function ?
          GetPointedToBy(IdType, NoKey) :
          IdType,
      OilErrorType()));
}
```

This macro is invoked in definition 292.

defkeyfunc.h[291] ≡

```
#ifndef DEFKEYFUNC_H
#define DEFKEYFUNC_H

#include "eliproto.h"
#include "deftbl.h"
#include "buildtype.h"
#include "envmod.h"

tOilTypeSet GetTypeWhenIdUseIsaFunction ELI_ARG((DefTableKey, CoordPtr));
tOilTypeSet GetTypeWhenIdUseIsDefined ELI_ARG((DefTableKey, CoordPtr));

#endif
```

This macro is attached to a product file.

defkeyfunc.c[292] ≡

```
#include "defkeyfunc.h"
#include "err.h"
```

```
#include "buildtype.h"
#include "type.h"

#include <stdio.h>
```

Guarantee DefTableKey for function[289]
Guarantee declaration for DefTableKey[290]

This macro is attached to a product file.

11.3 Specification files

compute.lido[293] ≡

Rule computations[278]
Required types[281]
Miscellaneous Types[279]
Shut Up[280]
Unary Indication rules[284]
Binary Indication rules[285]
Get return type of function_definition[282]
Sanity Checks for constant values[286]
Sanity checks for expression types[288]

This macro is attached to a product file.

The following line is required to make ELI find the Oil header file with all of the Oil definitions in it.

compute.specs[294] ≡

```
$/oil/oiladt2.h
$/Adt/csm.h
```

This macro is attached to a product file.

compute.head[295] ≡

```
#include "defkeyfunc.h"
```

This macro is attached to a product file.

Another attribute of a `DefTableKey` is an oil type, which is used when the `DefTableKey` refers to a type.

compute.pdl[296] ≡

```
"oiladt2.h"
OilOp         : tOilOp;
IsKnownConst: int;
```

This macro is attached to a product file.