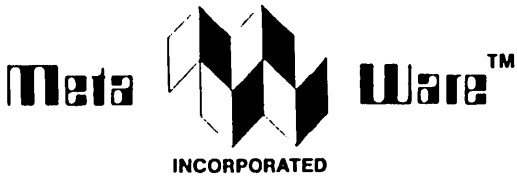


High C ™

Language Reference Manual

Version 1.2

by MetaWare™ Incorporated

High C ™

Language Reference Manual

Version 1.2

NOTICES

The software described in this manual is licensed, *not sold*. Use of the software constitutes agreement by the user with the terms and conditions of the End-User License Agreement packaged with the software. Read the Agreement carefully. Use in violation of the Agreement or without paying the license fee is unlawful.

Every effort has been made to make this manual as accurate as possible. However, MetaWare Incorporated shall have no liability or responsibility to any person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this manual, including but not limited to any interruption of service, loss of business or anticipated profits, and all direct, indirect, and consequential damages resulting from the use of this manual and the software that it describes.

MetaWare Incorporated reserves the right to change the specifications and characteristics of the software described in this manual, from time to time, without notice to users. Users of this manual should read the file named "README" contained on the distribution media for current information as to changes in files and characteristics, and bugs discovered in the software. Like all computer software this program is susceptible to unknown and undiscovered bugs. These will be corrected as soon as reasonably possible but cannot be anticipated or eliminated entirely. Use of the software is subject to the warranty provisions contained in the License Agreement.

A. M. D. G.

Trademark Acknowledgments

The term(s)	is a trademark of
High C, MetaWare	MetaWare Incorporated
MS-DOS	Microsoft Corporation (registered tm.)
Professional Pascal	MetaWare Incorporated
UNIX	AT&T Bell Laboratories

Feedback, Please

(Upon first reading.)

We would greatly appreciate your ideas regarding improvement of the language, its compiler, and its documentation. Please take time to mark up the manual on your ***first*** reading and make corresponding notes on this page (front and back) and on additional sheets as necessary. Then mail the results to:

MetaWare™ Incorporated
412 Liberty Street
Santa Cruz, CA 95060

MetaWare may use or distribute any information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use that information. If you wish a reply, please provide your name and address. **Thank you** in advance, The Authors.

Page Comment

Feedback, Please

Page Comment

Temporarily, this manual has been printed via dot matrix rather than typeset.

The manual is expected to be typeset soon, after some feedback comes in from early customers.

Since such things often get delayed, please be sure to send in your feedback as soon as possible — if your suggestions do not get in the first typeset version, they may have an effect on the next.

Thank you for your patience, understanding, and suggestions.

The folks at MetaWare.

Contents page(s)

for High C ™ Language Reference Manual . *total 234 pp.*

Contents page(s)

Contents page(s)

Contents page(s)

Contents page(s)

1

Introduction

1.1 Scope, Audience, Purpose

This is a language reference manual for C. It does not attempt to teach C programming to those unfamiliar with the language. For an informal introduction to C, consult Kernighan and Ritchie [K&R].

The writing of this manual was prompted by the lack of any precise description of C. A common way to answer a question about C is to "see what the compiler does". This is anathema to writing programs, portable or not. Clearly C has suffered from being partly defined, then implemented. The original definition [K&R] is quite incomplete.

Our goal here is to provide a single document that answers all machine-independent questions about C. To that end we use formal notation, such as context-free grammars, where possible to avoid ambiguities. Thus, the reader of this manual must have a tolerance for formality, but hopefully will be rewarded by always getting answers.

1.2 Need for Formality

To illustrate a question poorly addressed in the literature, consider this C program fragment:

```
struct s *p;                    /* Declaration A.      */
int f() {
    struct s {int x, y;};       /* Declaration B.      */
    p->x = 1;                   /* Reference R.        */
    }
int g() {
    p->x = 1;                   /* Reference S.        */
    }
```

Is this a legal C program? The issue at hand is whether declaration B "completes" declaration A by supplying the fields x and y. The original definition of C [K&R] does not say how A and B relate, if at all. Another recent book on C [H&S] says that "a structure type reference [may] precede the corresponding type definition (provided the reference occurs in a context where the size of the structure is not required)". Is definition A above a type reference preceding the corresponding type definition B?

For at least one compiler [4.2BSD] the answer is yes, so that within function f reference R is permitted. However, after the closing brace of f, the completion B is somehow lost so that reference S is illegal. Yet declaration A can no longer be completed: supplying another completion draws an error diagnostic.

No published C reference known to us properly addresses this issue. The answer we provide is that a structure declaration can complete a previous one only if both are "declared at the same level", a notion that we make precise.

1.3 Which C?

When we claim to give a language definition for C, a question arises: "Which C?". The original definition of C [K&R] did not give C a complete definition. Compilers for C differ where K&R is silent or obscure.

In some sense we are defining what C "should be" or "should have been defined as". In doing so we are not defining an entirely new language, including in it all of our favorite constructs. Instead, we observed what existing C compilers for C do with C's ill-defined spots, and produced a definition that attempts to be faithful to the most rational decisions made in those compilers.

Furthermore we took into consideration the C language draft standard produced by the ANSI C standardization committee X3J11 (and have participated in the development of

that document). The X3J11 standard is another C definition, although less formal than ours, that tries to clarify the obscure and define the ill-defined, in addition to extending C where the K&R language is perceived as being weak.

Therefore this document should on the whole be compatible with most C compilers, especially with those written with attention paid to X3J11's work. We also point out where our definition differs from that typically implemented by popular compilers, or where it differs from the X3J11 work.

A few of the language features we describe exist in no other C compiler or definition or in the X3J11 work. These features are particular to MetaWare's own High C language. Most are simple changes, and are included in the main body of the language definition; those that are not simple are relegated to Appendix *Language Extensions*. All extensions, however, including those originating in X3J11, are listed in that appendix and the reader is invited to consult it for a summary.

1.4 Exclusions

For the purposes of this manual, the C library is not considered part of the C language and is not treated here. The ANSI standardization of C includes a library standard along with a language standard, but also recognizes so-called "freestanding environments" that do not include any library.

1.5 Format

[Syntax, Constraints, Semantics, Discussion, Machine Dependencies]

In this manual each C construct is presented in several sections:

Syntax gives a regular-right-part context-free grammar that describes the syntactic structure or "form" of the construct. Section *Notation* explains grammars and how syntax is divided into two levels: lexicon and phrase structure.

Constraints lists rules that each instance of the construct must obey statically if it is to be well-formed. Such rules are in addition to the requirements imposed by the grammar, and generally are rules that cannot be described by a context-free grammar.

Semantics states the meaning of a well-formed construct; e.g. what happens at run-time: what value or effect it has.

Discussion describes differences, extensions, and restrictions of this definition as compared to other C language definitions [K&R] [X3J11], or as compared to a popular implementation [4.2BSD]. Ramifications of the language definition that might otherwise go unnoticed are noted here. Occasionally we include examples and/or a brief summary description of the construct, especially when such is useful to illuminate why the construct was put in the language the way it was.

Machine Dependencies
describes aspects of the construct that are particular to certain machines.

Any of these sections may be omitted, as appropriate. For example, if section *"Machine Dependencies"* is omitted, one may infer that there are no known or relevant machine dependencies for the particular construct. Occasionally, sections that would otherwise be omitted are included for emphasis.

Material that should be included in *Discussion* may instead be included in the other categories, when moving it to *Discussion* would place the material too far out of context. Such material is flagged by the brackets ***Note*** and ***End of Note***. We emphasize that such material is commentary only and not necessary to the language definition.

Examples are generally started by the text ***Example***. When it may be unclear where the end of the example is, the text ***End of Example*** is used.

Some authors prefer to use the phrases "Context-free Syntax" and "Context-Sensitive Syntax" instead of "Syntax" and "Constraints". Still others prefer "Static Semantics" to "Constraints" and "Dynamic Semantics" to "Semantics". Rather than take sides we have used different terms.

1.6 Key Words and Phrases

[sample; word or phrase]

Following most subsection headers is a line such as the latter containing key words and phrases related to that subsection. They give a quick idea of what the section is about. The Index lists each word or phrase alphabetically by word or each word in a phrase.

1.7 References

[K&R] Kernighan, Brian W. and Dennis M. Ritchie: **The C Programming Language**. Prentice-Hall, Inc. Englewood Cliffs, NJ 07632, 1978.

[H&S] Harbison, Samuel P. and Guy L. Steele, Jr.: **C: A Reference Manual**. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1984.

[X3J11] American National Standards Institute X3J11 committee on the standardization of the C programming language. X3 Secretariat, CBEMA, 311 First Street NW, Suite 500, Washington, DC 20001. The draft document from which we cite differences is dated April 30, 1985, document 85-045, and is available from CBEMA at the above address.

[4.2BSD] The portable C compiler as it exists on Berkeley's 4.2 distribution of the UNIX operating system on Digital Equipment Corporation's VAX computers.

In the sequel, the following abbreviations hold: KR for [K&R], X3J11 for [X3J11], and 4.2BSD for [4.2BSD].

2

Notation

2.1 Lexicon versus Phrase-Structure

[lexical versus phrase-structure syntax; words, texts; constraints; context-free grammar; regular expressions]

Syntax can be usefully broken into two parts: lexicon and phrase-structure.

Lexical syntax refers to the valid sequencing of characters in a program to form words, and the naming of those words.

Phrase-structure syntax refers to the valid sequencing of the words to form phrases.

For example, consider the C program

```
main() {int i = 1;}
```

On the lexical level, the program appears as the character sequence

```
'm', 'a', 'i', 'n', '(', ')', ' ', '{', 'i', 'n', 't',
' ', 'i', ' ', '=', ' ', '1', ';', '}'.
```

The lexical syntax, as will be seen, permits such a character sequence and specifies the corresponding word names and their associated sequences of characters, or *texts*, indicated next:

Word name	*Text*
<IDENTIFIER>	'm','a','i','n'
(	(none)
)	(none)
{	(none)
int	(none)
<IDENTIFIER>	'i'
=	(none)
<INTEGER>	'1'
;	(none)
}	(none)

At the phrase-structure level, the sequence of words above, namely,

```
<IDENTIFIER> ( ) { int <IDENTIFIER> = <INTEGER> ; }
```

is a valid C program. Hence the original character sequence forms a valid C program, at least on the lexical and phrase-structure levels. Note that some of the words illustrated above have no text. This is by design and not by any restrictions inherent in the formalisms used here.

Phrase-structure description is concerned only with the names of words, never with texts (with one exception in Section *Preprocessor*). For example, C's phrase structure specifies that an <IDENTIFIER> be the name of a function, not, say, an <INTEGER>; but which <IDENTIFIER> in particular is not a phrase-structure concern.

On the other hand, the constraints of C — the context-sensitive syntax — are concerned with the texts of <IDENTIFIER>s. For example, a usage of an <IDENTIFIER> with text T must generally follow a declaration of an <IDENTIFIER> with text T.

C's lexicon and phrase structure are each formally defined here by a context-free grammar.

Grammar illustrations. Illustrations from each kind of grammar are appropriate before proceeding further.

Consider a fragment from the lexical grammar:

```
Identifier-> Letter (Letter|Digit)*    =>'<IDENTIFIER>';
  Letter   -> 'A'..'Z' | 'a'..'z' | '_' ;
  Digit    -> '0'..'9' ;
```

The "=>'<IDENTIFIER>'" specifies that a word is to be named <IDENTIFIER>. The text for that word is a letter followed by zero or more letters and digits, where a "letter" includes the underscore ('_') character.

According to the lexical grammar, words can generally appear in any order, as indicated by the following lexical grammar fragment near the top of the grammar:

```
Words  -> Word*;
Word   -> String | Char | Number | Identifier
        | Delimiter | Punctuator | Operator | Comment;
```

As an illustration from the phrase-structure grammar, consider the rule

```
Statement
-> 'for' '(' First:EL?
         ';' Next: EL?
         ';' Last: EL?
         ')' Body: Statement
```

It describes the C **for** statement, consisting of:

the words **for** and **(**,

(optionally) any sequence of words generated by the nonterminal EL, the word **;**,

(optionally) any sequence of words generated by the nonterminal EL, the word **;**,

(optionally) any sequence of words generated by the nonterminal EL, the word **)**,

and finally any sequence of words generated by the nonterminal Statement.

2.2 Grammar Notation

[context-free grammar; regular expressions: list, "*", "+", "?", "|", ".."; adjectives; <DELETE>, <AS_IS>; reserved word]

The context-free grammars used here contain *regular expressions*, which are expressions composed with postfix (*, +, ?) and infix (**list**, |, ..) operators having the following meanings:

Expression	*means*
X*	zero or more Xs.
X+	one or more Xs.
X?	zero or one X, i.e. X is optional.
X|Y	either X or Y.
X **list** Y	one or more Xs, separated by single occurrences of Y; equivalent to X (Y X)* and (X Y)* X, giving X, X Y X, X Y X Y X, etc.
X..Y	the sequence of characters from X to Y, inclusive (meaningful only in lexical grammars).

*, |, ?, and .. were used in the grammar examples above.

All terminal symbols, e.g. **'for'** and **'_'**, are single-quoted in grammars to avoid ambiguity. Nonterminals are not quoted, e.g. Statement and Letter.

Parentheses of the forms (...) and <...> in grammars override precedence. In a lexical grammar the operator => specifies the name of a word; the name is the string following the =>.

So-called *adjectives* in grammars describe phrases of the grammar. The practice is borrowed from the Ada reference manual, where adjectives are typeset in italics. For example, *Static*_Expression in Ada is really the nonterminal Expression, but with the "reminder" that the Expression must be "static".

Adjectives are purely commentary in grammars, but often an adjective is mirrored in a constraint. Here we employ normally-typeset (non-italicized) identifiers followed by a ":" to denote adjectives. Adjectives appeared in the phrase-structure example above; *viz.* First, Next, Last, and Body.

Grammars may contain comments, which begin with "#" and continue to the end of the line.

Special words. The words <DELETE>, <AS_IS>, and <IDENTIFIER> have special meanings in lexical grammars, as follows:

A word named <DELETE> is not to be considered in phrase-structure or any other analysis. <DELETE> is used so that comments and so-called "whitespace" need not clutter the phrase-structure grammar. Effectively, lexical analysis deletes such words.

A word named <AS_IS> with text T is to be instead considered the word named T, with no text. For example, the word <AS_IS> with text (is instead the word (. This device is employed in naming operator and punctuation symbols.

Finally, any word named <IDENTIFIER> with text T is to be instead considered a word named T if T appears in the phrase-structure grammar. For example, the <IDENTIFIER> word with text 'f' 'o' 'r' is to be considered instead the word 'for', which appears in the phrase-structure grammar. These reserved <IDENTIFIER>s, called *reserved words*, are thus distinguished from ordinary <IDENTIFIER>s.

Additional notation. We defer explaining some rarely-used grammar constructs to the sections where they are used.

2.3 Lexical Ambiguity

The lexical grammar is ambiguous, for economy of expression. For example, the characters 'A', 'B', '1' can be interpreted as the word <IDENTIFIER> with text 'A', 'B', '1' or the two <IDENTIFIER>s 'A' and 'B' and the <INTEGER> '1'. In all cases ambiguity is resolved in favor of the longest possible word.

2.4 Program Text Conventions

C program fragments in running text are double-quoted, and reserved words are boldfaced. For example: "**int** x, y;" In addition, C text is always reproduced in fixed-width font rather than varying-width font. Examples apart from running text are not quoted.

2.5 Constraints and Semantics

The specification of constraints and semantics is keyed to the nonterminals and adjectives in the grammar. Consider the phrase-structure grammar fragment:

```
External_declaration
-> Unspecified_declaration:
   ( Function_definition
   | Non_function_definitions
   )
-> Specified_declaration          # With specifiers.
-> ';'                            # Allowed by KR.
```

Specifications may be keyed to any of the nonterminals Specified_declaration, Function_definition, Non_function_definitions, or the adjective Unspecified_declaration, which "modifies" the alternation (|). As mentioned before, Adjectives do *not* figure in the formal grammatical definition of the context-free syntax.

2.6 Section References

References throughout the text from one section to other sections are made by section and subsection title only, except when subsection references are made within a section, where the subsection number is most often used. A '/' separates a section title from a subsection title, when the latter is included. For example, Section *Concepts/Lifetimes* refers to Section *Concepts*, Subsection *Lifetimes*. The location of each section and subsection can be found in the Table of Contents.

2.7 Composition of a C Program

[source files; compilation unit; linking; program execution; independent translation; preprocessor]

A complete C program P consists of a collection of declarations such that there exists exactly one definition of a function whose name is "`main`". P's execution begins with this function.

P need not be translated all at once. Typically, components of P are kept in separate *source files* that are independently translated. We assume the notion of "source file" is atomic and therefore provide no definition for it here.

A *compilation unit* is a single source file F unless F contains preprocessor directives that specify the inclusion of other source files as part of the unit. We leave unspecified exactly how compilation units are determined, for that is, in general, a host-environment-dependent concept not of concern here.

The focus here is only on the semantics of compilation units and the semantics of the combined translated results of separately compiled units. We shall not discuss the process of *linking* which is typically used to prepare the translated results for "execution", nor the execution process. The translated linked result is a C program that may be *executed* i.e. its meaning made manifest.

The reason that separate translation must be addressed is that the semantics of a C program when independently translated are not identical to those when the components are "glued" together in a single source file and translated all at once. For example, in each of two separate compilation units, the declaration "`static int x;`" may appear; however, two such declarations cannot appear in a single unit.

2.8 When is a Program a Program: the Preprocessor

[conditional compilation; file inclusion; macro replacement]

A "normal" description of a programming language l defines lexical syntax, phrase-structure syntax, constraints, and semantics. Text P is a legal program in L (and therefore has meaningful semantics) when: (a) P is lexically valid; (b) the word sequence defined by the lexical syntax forms proper phrases; and (c) the constraints are satisfied. Thus, a program's validity can be determined without reference to any "processors" that do syntax checking or constraint analysis, such as "scanners", "parsers", and "constrainers".

For C the description cannot be so simple. Part of the definition of C includes a so-called *preprocessor* that modifies the word sequence defined by the lexical syntax before that word sequence is subjected to phrase-structure analysis. The preprocessor is complex and interacts with the lexical analysis in such a way that the only reasonable way to understand whether a program is valid is to simulate the machinations of the processor, at least through phrase-structure analysis.

Therefore, for the purposes of this document, text P is a legal C program if: (a) P is lexically valid; (b) the words *as transformed* by the preprocessor form proper phrases; and (c) the constraints are satisfied.

The C preprocessor effects: (a) *conditional compilation*, including and excluding program texts from compilation based on the evaluation of Boolean expressions; (b) *file inclusion*, logically substituting for file inclusion directives the contents of source files referred to in the directives; and (c) *macro replacement*, the transformation of some sequences of words into other sequences of words.

The preprocessor is explained in more detail in Section *Preprocessor*.

The term "preprocessor" is due to its often being a program separate from the C compiler per se, transforming the source file before the C compiler analyzes it. However, the

definition of the preprocessor component of the language does not preclude its incorporation into the lexical analysis phase of a C compiler, and we know of at least one compiler that does so, namely the MetaWare High C ™ Compiler.

3

Concepts

The concepts described in this section are needed for the later description of C relative to its syntax.

Defined concepts are illustrated by example C programs, even though the syntax of C programs has not been discussed yet. Readers familiar with C will find the examples helpful; those not familiar can make good use of the examples upon second reading, after studying the syntax of C.

3.1 Name Spaces

[declaration property set; ordinary, tag, label, and struct and union name spaces]

Every declaration in C associates a name with a *declaration property set* within a specific *declaration name space.*

A name space can be viewed as a mapping taking a name into its declaration property set. There are three distinguished name spaces in C: the *ordinary* name space, the *tag* name space, and the *label* name space. There is also a name space for each distinct **struct** and **union** type, which contains the type's member names.

The explanation of each declaration gives the name space of the name declared. The three spaces permit the same name to be associated with up to three different property sets. The syntactic context of a name always determines which name space holds the association. For example, a label name either precedes a ":" at the beginning of a statement, or follows the reserved word **goto**; it appears nowhere else.

The elements of declaration property sets are described later in this section.

Examples:

```
int x;              /* x: ordinary name space.                */
struct s {          /* s: tag name space.                     */
   int x;           /* x: name space of the struct type s.    */
   } y;             /* y: ordinary name space.                */
L: goto L;          /* L: label name space.                   */
```

3.2 Blocks, Origins, Defining Points, and Scopes

[duplicate declaration]

The discussion in this subsection does *not* apply to the member names of **struct** and **union** types; the issues of blocks, origins, defining points, and scopes are irrelevant to such names.

Every declaration is contained in one or more *blocks*, which are regions of program text. The specific locations of such regions are not described here; the description of a construct having an associated block contains the description of that block.

Generally, a block Inner can be a part of another block Outer, in which case Outer is said to *contain* Inner. The block Inner is then called an *inner block* of Outer, while Outer is called a *surrounding block* of Inner. In general a program consists of a hierarchy of nested blocks.

The innermost block in which a declaration occurs is called the *origin* of that declaration.

The *defining point* of a name is the occurrence of the name in its declaration.

Every declaration D has a *scope*. The scope of D is that program text in which the declared name N is associated with the property set of D. The scope of D is generally the program text extending from the defining point of N to the end of D's origin, but there are exceptions. The first is when the declared name is re-declared in a contained block and in the same name space:

Suppose that block Outer contains block Inner, and that Outer is the origin of a declaration D of name N within name space S. If Inner is the origin of another declaration D' of N within S, then the program text from the defining point of N in D' to the end of Inner is excluded from the scope of D.

Other exceptions to this rule are documented with the exceptional constructs.

If the scope of two distinct declarations D and D' of a name N in the same name space overlap, this is known as a *duplicate declaration of* N. Such duplicate declarations are prohibited. This means, for example, that D and D' may not appear in the same block; e.g. "`int a, a;`" is not allowed.

However, two declarations of the same name in different name spaces is permitted: "`struct s {int y;} *p; int s;`", for example — the first s is in the tag name space and the second is in the ordinary name space.

Under certain circumstances, apparent duplicate declarations are permitted. In addition, certain combinations of non-duplicate declarations are prohibited. These situations are described in Subsection *Independent Translation* below.

Examples:

```
(a) int x;                  /* x:  ordinary name space.                */
(b) struct x {              /* x:  different name space than in (a);   */
                            /*     hence not a duplicate declaration.  */
(c)     int x;              /* x:  issues of scope, block, etc.        */
                            /*     do not apply to this name.          */
(d)     struct x {...} y;   /* x:  conflicts with (b): duplicate dcln. */
(e)     } x;                /* x:  conflicts with (a): duplicate dcln. */
(f) x:  goto x;             /* x:  different name space than in (a)    */
                            /*     or (b); hence not a duplicate dcln. */
```

3.3 Declaration Property Sets

```
[mode, type, storage class; modes var, value, fcn, typedef; modes
struct-tag, union-tag, enum-tag; modes member, field; lvalue, rvalue]
```

A declaration property set contains the information about a name gleaned from its declaration. This set prescribes how the name may be used subsequently in its declaration's scope. This set is not something a C program manipulates, but rather is needed by the reader of a program to understand it in detail. The set may be used by a C language processor to check constraints and perform translation.

The property set consists of up to three "attributes": a *mode*, a *type*, and a *storage class*. Generally the mode and type attributes are typeset in lowercase **boldface**. Also, an expression has mode and type (but no storage class), even though it does not interact with the name spaces. Names in the label name space do not have a mode (it is unnecessary), but names in the other name spaces have all three attributes.

The mode is one of **var**, **value**, **fcn**, **struct-tag**, **union-tag**, **enum-tag**, **member**, **field**, and **typedef**. Modes are used to capture the ways in which names can and cannot be used. For example, the mode **field** is used to prevent taking the address of a structure bit-field. The mode of a name or expression E may be interpreted as follows:

Mode	*means*
In the ordinary name space:	
var	E may be used on the left side of an assignment.
value	E is legal only on the right side of an assignment.
fcn	E denotes the entry point of a function.
typedef	E is a typedef.
In the tag name space:	
struct-tag	E is a **struct** tag.
union-tag	E is a **union** tag.
enum-tag	E is an enumeration tag.
In the name space for a **struct/union** *type:*	
member	E is a non-field member of a structure or union.
field	E is a field of a structure or union.

These explanations are deliberately incomplete here; the full meaning of the modes is meant to be obtained from reading the sections that attribute modes to expressions and names and the sections that require certain modes. KR uses the terms "lvalue" and "rvalue" as modes; in our notation, rvalue is **value** and lvalue is the the union of the modes **var** and **field**.

Types and storage classes are discussed in the remainder of this chapter.

If a declaration associates a name N with a property set containing mode M, type T, and storage class C and within name space S, we also say that the declaration "declares N to be of mode M, of type T, of storage class C, and within S". We may omit any of M, T, C, or S, as appropriate.

Examples:

```
int x, f();          /* x: mode var; f: mode fcn.                  */
typedef
   struct s {        /* s: mode struct-tag.                        */
      int x;             /* x: mode member.                       */
      int z:3;           /* z: mode field.                        */
      } y;           /* y: mode typedef.                           */
enum e {a,b};        /* a,b: mode value; e: mode  enum-tag.        */
union u {int x;};    /* x: mode member;  u: mode union-tag.        */
L: goto L;           /* L: no mode necessary.                      */
```

3.4 Values, Types, and Objects

[variable; type denotation; size of object; integral, floating, arithmetic types; Signed-Int, Unsigned-Int; Signed-Short-Int, Unsigned-Short-Int; <CHAR>; Signed-Char, Unsigned-Char; Signed-Long-Int, Unsigned-Long-Int; Float, Double, Long-Double; Void]

Values are entities upon which operations may be performed.

Types classify values according to the operations that may be performed upon them: a type is a set of values.

Objects can "contain" values, usually of a single type; at different times during the execution of a program an object may hold different values.

Declarations in C almost always declare objects. Some objects may have other objects as components. As in other languages, when we speak of a *variable* we mean a declared name denoting an object, since the variable's contents ca vary. In contrast, a declared name denoting a function does not denote an object and is not spoken of as a variable.

Since a type is a set of abstract values, we can never write down a type in this document, but merely use a word or words to *denote* a type. For example, we use "Signed-Int" to denote the type consisting of a subset of all the signed integers. (The particular subset is implementation-defined.)

Values and objects of a given type T are represented in a fixed number of storage units; this number is the *size* of T.

The storage unit must be able to contain any value of type Signed-Char or Unsigned-Char and is usually the eight-bit byte. It must be possible to express the address of each storage unit in the target architecture; these addresses are the values of pointer types (described below).

Basic types. There is a set of basic types in C, and methods for denoting new types in terms of basic types. The Table below lists the names for the *basic types.* The two distinct subsets, *integral* and *floating*, comprise the *arithmetic types:*

Table Arithmetic types.

Integral types:		*Floating types:*	*Other:*
Signed-Int	Unsigned-Int	Float	Void
Signed-Short-Int	Unsigned-Short-Int	Double	
Signed-Long-Int	Unsigned-Long-Int	Long-Double	
Signed-Char	Unsigned-Char		

Shorthand. Without some shorthand, the phrase "denoted by" will appear all too often in this document. For example, consider the precise but cumbersome phraseology: "If T1 is the type denoted by Unsigned-Int and T2 the type denoted by Signed-Int, then T3 is the type denoted by Unsigned-Int".

Hereafter we drop the "denoted by" when the intent is clear, and use the denotations as if they were the actual types.

After all, the denotations are just the *names* for the types, and in common English usage we do not introduce "the man denoted by Fred" — rather, we introduce Fred.

Nevertheless, the distinction between types and their specification in a C program must be kept clear. In a C program one writes **"int"** to denote the type Signed-Int. However, what one writes *is not* the type, but merely denotes it.

The difference shows up better in the C phrase **"int short unsigned"**: this phrase and in fact any permutation of those three words (and possibly intermixed with a Storage_class) denotes the type Unsigned-Short-Int.

Types, denotations. Because C syntax permits more than one way of denoting some types, we have chosen our own type denotations; we use exactly one denotation for each type. How to denote types in C is not detailed until the next section. Here we discuss briefly the repertoire of basic types.

There are no values of type Void. Void is used primarily as the return type of a function returning nothing.

The type Signed-Int corresponds to a signed integer. Typically, its size is "natural" to the machine's arithmetic abilities.

Signed-Short-Int and Signed-Long-Int are two other signed integer types. The set of values of a Signed-Short-Int is a subset of that of a Signed-Int, which is a subset of that of a Signed-Long-Int. These variations on Signed-Ints are provided to obtain more efficient or larger integer calculations where necessary. Even if an implementation defines Short-Int or Long-Int to have the same set of values as Signed-Int, all three are nevertheless distinct types.

Each of the three Signed-Int types has a corresponding unsigned type whose size is the same as the corresponding signed type. The set of non-negative values of a signed type is a subset of that of its corresponding unsigned type.

The values of a Signed-Char are a subset of the values of a Signed-Int. The values of an Unsigned-Char are a subset of the values of an Unsigned-Int. However, the values of a Signed-Char need not be a subset of the values of a Signed-Short-Int; likewise with the corresponding unsigned types. Essentially, the Char and Short-Int types are two not necessarily related integer types shorter than Signed-Int.

The rules for assignment compatibility presented later in this Section require two distinct integral types to have the same set of values if the two types are of the same size and of the same "signedness".

An object declared of type Signed-Char can hold any <CHAR>. <CHAR>s specified without the '\ddd' or '\xddd' form are guaranteed to be non-negative; otherwise, the sign of a <CHAR> is implementation-defined. Unsigned-Char can hold any non-negative <CHAR>.

Float, Double, and Long-Double are floating(-point) types. Any Float value is representable in Double. Any Double value is representable in Long-Double. Even if an implementation defines Float and Double to have the same set of values, they are nevertheless distinct types; the same holds for Double and Long-Double.

The set of integral values is not necessarily contained in the set of floating values, and vice-versa.

3.5 Denoting New Types

[constructed types; array and component type; incomplete types; pointer type; structure and union types; member-list; scalar and aggregate types; functionality types; prototype and non-prototype functionalities; parameter types; type notation]

There are methods for denoting new, non-basic types based on existing types; we call these the *constructed types.* The C syntax for denoting these types is given in a later section; here we specify our own notation to denote such types.

If I is an integral value and T any type, [I]:T is an *array type* with *component type* T, and [?]:T is a *incomplete array type*

with *component type* T. An incomplete array type is one for which the size of the array is unknown.

If T is a type, *T is a *pointer type* with *base type* T.

If M is a Member_list (see Section *Declarations/Tagged Types* for its syntactic definition), Struct{M} is a *structure type* with *member-list* M, and Union{M} is a *union type* with *member-list* M. The C language also provides for *incomplete* Struct and Union types, where the Member_list is not given. Such types are denoted by Struct{?} and Union{?}, respectively.

The types [?]:T (for any T), Struct{?}, and Union{?} are collectively referred to as the *incomplete* types. Incomplete types have fewer uses than complete types. Primarily, objects of incomplete types may not be declared, and the sizes of incomplete types are specifically undefined.

Arithmetic types and pointer types are collectively called *scalar* types, and array, **struct**, and **union** types *aggregate* types.

Examples:

```
extern int a[][3];      /* a:   type [?]:[3]:Signed_int.      */
double *pd;             /* pd:  type *Double.                 */
double *apd[3];         /* apd: type [3]:*Double.             */
double (*pad)[3];       /* pad: type *[3]:Double.             */
struct {int x;} *aps[3];
                        /* aps: type [3]:*Struct{int x;}.     */
```

Functionality types. There are four kinds of *functionality* types: those types describing functions. The four kinds fall into two classes: the *prototype* functionalities and the *non-prototype* functionalities:

(a) (?) -> T — for T a type;
(b) (T_1,...T_n) -> T
(c) (T_1,...T_n)$_p$ -> T — for T, T_1,...T_n types, $n \geq 0$.
(d) (T_1,...T_n,...)$_p$ -> T

The first two kinds represent notions in KR C, and are the non-prototype functionalities. In class (a) the function returns

values of type T, but the number and types of its parameters are unknown. In class (b) the parameter number and types are known: $T_1,...T_n$ (if n = 0, there are no parameters). (b) can arise only from a function definition, never from a declaration alone, since any attempt to specify parameter types in a declaration alone requires the use of prototype functionalities. See Subsection 3.10 for the distinction between definitions and declarations.

The last two kinds are the prototype functionalities. In class (c) the parameter number and types are known. In class (d) the trailing "..." indicates that more arguments may be passed than there are parameters declared.

Prototypes are a recent (ANSI) addition to C. They allow more secure Pascal-style argument type checking at function calls. The constraints and semantics of calls to functions of prototype functionality are quite different from those without such; see Section *Expressions/Function Call*. A declared function f has a prototype functionality when the nonterminal Abstract_parameters is used to specify f's parameters; it has non-prototype functionality otherwise (see Section *Declarations/Function Definition*).

Examples:

```
int f1();           /* f1: type (?) -> Signed_int.              */
int f2(void);       /* f2: type ()p -> Signed_int.              */
double f3(unsigned char c, float f);
                    /* f3: type (Unsigned_char,Float)p -> Double.  */
double f4(unsigned char c, float f){
   ...              /* f4: same as type of f3.                  */
   }
double f5(c, f) unsigned char c; float f;{
   ...              /* f5: type (Unsigned_char,Float) -> Double.  */
   }                /* Not same type as that of f3:  not a prototype. */
int *(*f6(double))[3];
                    /* f6: type (Double)p -> *[3]:*Signed_int.  */
```

Note that compatibility with old C demands rather awkward notation to declare an external function taking no parameters: "**(void)**", as in f2 above. This is because

"()" alone declares a function whose parameter types are unspecified. Declarations like that of f1 should be avoided.

Why new type notation. We introduced this new notation have an accurate way of referring to types. The syntax used in C to specify types is not conducive to easy comprehension of the specified type, nor is it appropriate to the specification of language rules for types. This new notation borrows symbols from C to facilitate readability.

This new notation also has an advantage over C's notation for denoting types in that it requires no precedence or associativity rules to understand, nor therefore parentheses to override precedence and associativity, and it is unambiguous (a grammar for it, which is implicit in the above rules, is in fact LR(0)). See the examples above, some of which contrast C's awkward need of parentheses with the clean parenthesis-free type notation.

Type "expressions" constructed with this notation can be read strictly from left-to-right. For example, in the C declaration "`int *f();`", f has type ( )->*Signed-Int; in the C declaration "`int (*f)();`", f has type *( )->Signed-Int.

The rigor of the notation can yet be improved, since the definition of Struct and Union types relies on the syntactic notion `Member_list` with no further interpretation of the phrases of those categories. But the notation can be made completely formal only at considerable expense.

The tradeoff will be evident later when slightly more complex rules are necessary for defining type sameness and equivalence. We appeal to the reader's intuition that, for example, the text of the `Member_list` is insufficient to determine the full type; two `Member_lists` can be textually identical yet in different blocks so type references within the lists may be different.

3.6 Same Types

[constructed types; instance of construction]

Each instance in C of a constructed type denotes a type distinct from any other constructed type. Thus, although two type constructions in C may appear identical, they denote different types. It is as if each type had as a component its "instance of construction". Two types are the *same type* if they have the same instance of construction.

For convenience, our formal notation for types does not incorporate instance of construction, because once the notion of "same type" is formalized, the instance of construction concept will be used infrequently. However, a C language translator must encode the instance of construction in its internal representation of types.

Examples:

```
struct s {int x, y;} x1;
struct   {int y, z;} x2;
struct s x3;              /* A use, not a definition. */
char *x4, *x5;
int  x6[10], x7[10];
void V() {
   struct s {int x, y;} x1;
   }
```

There are three distinct declarations of Struct types; as specified in Section *Declarations/Tagged Types*, the third line references the type denoted by the constructor on the first line, so that x3 and x1 are of the same type. Even though the first and last declarations appear textually identical, they denote distinct types because they have different instances of construction. x4 and x5 are also of distinct types: two distinct *Signed-Char types. Similarly, x6 and x7 are of two distinct [10]:Signed-Int types.

3.7 Equivalent Types

[same variable; same types; similar types; independent translation]

The requirement of identical instance of construction in the concept of same type leaves a major problem unsolved: how to specify type security across independent translation. For example, rules explained later state that it is possible to declare a variable X in source file F1 and a variable X in file F2 and have them denote the same object, thus:

File F1:	*File F2:*
`int *X;`	`int *X;`

The two Xs are of distinct *Signed_Int types, since the instances of construction of the types are different. How can the "same object" have two distinct types?

The solution we propose is to require only that distinct declarations of the "same variable" be associated with "equivalent" types rather than the "same type". Two type denotations T and T' denote *equivalent* types if:

1. T and T' are of the form Struct{M}, and the corresponding types of each member in M are equivalent; or
2. T and T' are of the form Union{M}, and the corresponding types of each member in M are equivalent; or
3. T and T' are "similar" types.

Two types T and T' are *similar* if types S and S' are similar and

1. T and T' are the same type; or
2. T is of the form *S and T' of the form *S'; or
3. T is of the form [V]:S and T' of the form [V]:S'; or
4. T is of the form [?]:S and T' of the form [?]:S'; or
5. One is of the form (?) -> S and the other (T_1,...T_n) -> S'; or
6. T = I -> S and T' = I' -> S', and I and I' are of the same form, i.e. one of the parameter forms of the four functionality forms used in Subsection 3.5 above, and the corresponding parameter types are similar.

The reason for the "similar" subcategory of type equivalence is that similarity can be applied within a single C compilation unit, whereas the additional notion of type equivalence is used only for independent compilation.

Examples:

```
typedef char T;
struct  s {T x,y;} x1;
void V() {
   typedef int T;
   struct s  {T x,y;} x2;
   struct s2 {T x,y;} x3;
   }
char *x4, *x5;
int x6[10], x7[10];
int f1();
int f3(int x, int y);
int f3(x,y) int x,y; {...}
int f4(int a, int b) {...}
```

x1 and x2 are *not* of equivalent types: the two Struct{T x,y;} types are not equivalent — in one case T refers to the type Char and in the other the non-equivalent Signed-Int.

x2 and x3 are of equivalent types since the tags s and s2 do not figure in the type denoted by the declarations.

x4 and x5 are of distinct but equivalent (and similar) *Signed-Char types.

x6 and x7 are of distinct but equivalent (and similar) [10]:Signed-Int types.

f1 and f2 are not of equivalent types, but f1 and f3 are, and f2 and f4 are.

The KR definition of C does not recognize the problem of type security across independent translation. X3J11 is still deliberating it at the time of this writing. All C translators known to this author do not type-check across independently translated files, with attendant risk to the programmer if he provides inconsistent declarations.

3.8 Lifetimes

[global and local lifetimes; Compound_statement; storage classes]

Objects manipulated in a C program have two different kinds of "lifetimes": global and local.

An object of *global lifetime* exists and retains its value throughout the execution of the entire program.

An object of *local lifetime* is created upon each entry into the Compound_statement in which the object is declared and is discarded when the Compound_statement is exited. By "created" we mean only that the storage for the variable is allocated, but not that the variable is provided with any initial value(s). Storage is allocated even if the Compound_statement is entered "abnormally", through a jump transfer of control (e.g. **goto**).

The lifetime of an object is determined by its storage class, as discussed in the next subsection.

Examples:

```
int x;              /* x: global lifetime.                    */
void f() {          /* f: global lifetime.                    */
   int x;           /* x: local lifetime.                     */
      {
      int y;        /* y: local lifetime.                     */
   L: y = 3;        /* Storage for y is always allocated      */
      }             /* whenever we arrive at L,               */
   goto L;          /* even through means of this goto.       */
   }
```

3.9 Storage Classes

[automatic, static, and typedef storage classes; static-private, static-export, and static-import]

There are three storage classes: *automatic, static,* and *typedef.* The static storage class is further subdivided into *static-private, static-export,* and *static-import,* which are mutually exclusive and jointly exhaustive of static.

An object with the static storage class has global lifetime; an object with the automatic storage class has local lifetime.

There are no objects of the typedef storage class, and therefore no lifetime issues. This class exists only for definitional convenience.

Examples:

```
int x;              /* x: static-export  => global lifetime.   */
extern int z;       /* z: static-import  => global lifetime.   */
static void f(){    /* f: static-private => global lifetime.   */
   int x;           /* x: automatic      => local lifetime.    */
   }
typedef int T;      /* T: typedef    => lifetime irrelevant.   */
```

3.10 Declarations and Definitions

[type; storage class; storage allocation]

A declaration in C announces the properties of an identifier N, e.g. its type and storage class. Additionally, a declaration may be a *definition* of N; not all declarations are definitions.

Intuitively, definitions cause storage to be "allocated" for variables and the code body of functions to be specified. Exactly those declarations having storage class static-export, static-private, or automatic are definitions.

Examples:

```
int x;              /* x: static-export  => definition.        */
extern int z;       /* z: static-import  => not definition.    */
static void f(){    /* f: static-private => definition.        */
   int x;           /* x: automatic      => definition.        */
   }
typedef int T;      /* T: typedef        => not a definition,  */
                    /*      but irrelevant for typedefs since */
                    /*      no storage is allocated for T.    */
```

3.11 Independent Translation; Duplicate Declarations

[sharing declarations; separate compilation units; static-import, static-export, and static-private; storage-class; information similar; information increasing]

Generally, separate compilation units of a C program "share" declarations of data and functions — if there were no sharing, there would be no purpose for combining all the units into the single program. Sharing is achieved when two distinct declarations denote the same entity, as described in this subsection.

First, there are restrictions placed upon distinct declarations of the same name N appearing in distinct compilation units. Under the restrictions, the distinct declarations denote the same object or function and therefore achieve sharing among units.

Second, certain apparent duplicate declarations are permitted in a single compilation unit. "**extern int** a[]; ... **int** a[10];" exemplifies a common case, where the former declaration is typically contained in an included source file. We say "apparent" because the special rules described below specify that an apparent duplicate does not itself declare, but instead *modifies*, the previous declaration, so there is yet a single declaration. Therefore we can always speak of a single compilation unit having a single declaration for a name N in the outermost block.

The description of the duplicate declaration rules is involved. The reader may find it profitable to go through the rest of this manual in the frame of mind that a C program is a single compilation unit, then return here for multiple-unit issues.

Denotation of the same entity among separate compilation units. Consider all declarations in a C program of a given name N within the ordinary name space such that the storage class of each such declaration D_i is static-import or static-export. All the D_i denote the same entity if the following two conditions hold:

(a) the types specified in the D_i are equivalent, except that here type [?]:T is considered equivalent to [V]:T for any V, in any non-function declaration, and

(b) exactly one of the D_i declares N of storage class static export.

Note that this permits one or more static-private declarations of N in a program. Such declarations do not interact with other declarations of N of static storage class.

Examples:

Compilation Unit 1:

```
extern int N[];          /* Static-import.          */
extern int N[];          /* Static-import.          */
int N[2];                /* Static-export.          */
```

Compilation Unit 2:

```
extern int N[];          /* Static-import.          */
```

Compilation Unit 3:

```
static char N;           /* Static-private.         */
void F() {
   extern int N[];       /* Static-import.          */
   extern int N[];       /* Static-import.          */
   }
```

Among these three units there are only two distinct entities named N; one is declared of type Char in Unit 3 and the other is declared six times in the three compilation units. The following *additional* declarations are *not* allowed:

Compilation Unit 4:

```
void G() {
   extern float N; /*Static-import, but wrong type. */
   }
```

Compilation Unit 5:

```
int N;                   /* Duplicate static-export. */
```

Compilation Unit 6:

```
extern int N[3];        /* [3]:Signed_Int not equi- */
                        /* valent to [2]:Signed_Int. */
```

The single static-export declaration D may be viewed as the "seat" of the declared entity. The static-import declarations may be viewed as merely referencing D. (In fact, import and export declarations are commonly implemented this way.)

There may be at most one initialization specified for a set of multiple declarations. This restriction is imposed in Section *Declarations/Non-Function Definitions* on initializations where only static-export declarations are allowed to have initializations, and the rules here permit only one static-export declaration.

Note that the rules *require* a definition of storage class static-export for an identifier. Alternatively, some C language processors weaken rule (b) above to read

(b') at most one of the D_i declares N of storage class static-export.

These processors create the definition implicitly when the translated results for the independently translated compilation units are combined to form an executable program.

Rule (b) as it stands necessitates an aspect of program unreliability, in that at least two textually different declarations for the same name are required, one by the "provider" of the name (the definition declaration), and the other by any "user" of the name:

Provider:

```
int *X;
```

User:

```
extern int *X;
```

Changing X's type requires modifying two distinct declarations; if one modification is forgotten, the program is no longer correct. On the other hand, with rule (b'), the distinction between user and provider is only in the mind of the programmer, and both may include the identical declaration of X, so any change requires only the change of a single declaration:

Provider:	*User:*	*Shared declaration, in file "X.def":*
`#include "X.def"`	`#include "X.def"`	`int *X;`

Duplicate declarations in a single compilation unit. An apparent duplicate declaration D of a name N is allowed in the presence of a previous declaration D' of N in the same scope, provided certain conditions are satisfied. Furthermore D may "update" the declaration of D, i.e. revise its type or storage class so that from the defining point of N in D' through the the remainder of the scope of D, N is associated with a modified property set.

Example:

```
extern int *(*b[ ])[ ];
extern int *(*b[ ])[5];
       int *(*b[3])[ ];
```

The first line declares b of type []:*[]:*Signed-Int (an array of pointers to arrays of pointers to Signed-Ints). The second supplies the size of the second [], giving []:*[5]:*Signed-Int as the "updated" type of b. After the third declaration, b is of type [3]:*[5]:*Signed-Int, and its storage class is "changed" to static-export.

Here are the rules concerning would-be duplicate declarations. Assume declaration D declares N of mode M, type T, and storage class C, and apparent duplicate declaration D' declares N of mode M', type T', and storage class C'. D' is <u>not</u> a duplicate declaration of N if and only if either:

(a) M and M' are **var**, C and C' individually are static-import or static-export, and either T and T' are similar or T' is "information similar" to T (defined below); or

(b) M and M' are **fcn** and T and T' are similar. We further require that two distinct declarations of the same function must have identical parameter names, if given.

In case (a), if T' is "information increasing" — where information similarity holds yet T' has "more information"

than T, T' is "updated" to reflect the increased information. The increased information comes in the form of array bounds whose size are unspecified in T, i.e. the updating consists of changing an incomplete []:... type to a completed [V]:... type, where V is the new information. Furthermore, if C was static-import and C' static-export, C is changed to static-export, so that the net effect is that C is exported. Note that neither C nor C' can be static-private; e.g. "**extern int** a; **static int** a;" is illegal.

In case (b), if D' is not part of a function definition, it is essentially ignored. If D' is part of a function definition, it "replaces" the declaration D of N for the remainder of N's scope. See the examples at the end of this subsection.

Type T' is defined to be *information similar* to type T in the same way they are defined to be similar, except that in cases 2, 3, and 4, S' is information similar to S rather than just similar, and we need the addition of the following case 4':

4'. T is of the form [?]:S and T' of the form [V]:S'; in this case T' is said to have *more information* than T, *viz.* the array bound.

Example: Revisiting the example given above:

```
extern int *(*b[ ])[ ];
extern int *(*b[ ])[5];
       int *(*b[3])[ ];
```

the type of the second declaration is information-increasing with respect to the type T of b determined in the first declaration, so that T is updated to reflect the [5] bound. The type of the third is likewise increasing with respect to (the updated) T, so that T is updated again to reflect the [3] bound. Also, the storage class of b is altered to static-export.

Examples: (duplicate function declarations)

```
int f();              /* No parm information.                    */
int f(int);           /* Illegal: not similar to first.          */
int f(float r) {      /* Illegal: not similar to first.          */
   ...
   }
int g(int i);         /* Parm type and name given.               */
int g(float);         /* Illegal: float != int.                  */
int g(int j);         /* Illegal: j != i.                        */
int g(int i) {        /* Definition replaces first dcln of g.    */
   ...
   }
int h();              /* No parm information.                    */
int h(r) float r; {
   ...                /* Definition replaces first dcln of h.    */
   }
```

3.12 Compatible Types

[similar types]

Where two different types participate in an expression, they must sometimes be "compatible". See, for example, the ?: and == operators in Section *Expressions* and the definition of assignment compatibility below. Compatibility is like "similarity" except in allowing one exception: pointers to non-functions can be mixed with pointers to Void.

Two types are *compatible* if they are similar, or one is *Void and the other of the form *S, where S is not a functionality type.

Examples:

```
int  *x, *y; /* Types of x and y are compatible, since similar. */
void *z;     /* Types of x and z and of y and z are compatible. */
int  **ppi;
void **ppv; /* Types of ppi and ppv are not compatible.         */
```

3.13 Assignment Compatibility;Arithmetic Conversions

The notion of assignment compatibility is used to explain the constraints and semantics of the C assignment operators and function-call operator. In the context of those operators, arithmetic conversions can take place when certain types are not the same.

Intuitively, type R is assignment-compatible with type L if "$V_L = V_R$" is a valid assignment expression, where V_L is an object of type L and V_R is a value of type R. Below are the rules for assignment-compatibility.

The semantics of assignment permit potential conversion of V_R to V_R' in preparation for storing into V_L. In all cases except where R and L are integral types, the semantics of assignment are undefined when V_R' is a value not representable in type L. The semantics of converting V_R to V_R' are included in the rules below.

Constraints and Semantics

Type R is assignment-compatible with type L if one of these three rules apply:

1. R and L are the same type T, and T is not an incomplete type. *Semantics:* $V_R' = V_R$.
2. R and L are compatible pointer types. *Semantics:* $V_R' = V_R$.
3. R and L are distinct arithmetic types. *Semantics:*
   a. Both R and L are integral types.

      Let W_R be the width of R and W_L be the width of L. Let S_R be the sign (either "signed" or "unsigned") of R and S_L the sign of L. Consider the cases:

      $W_L > W_R$: the conversion truncates bits.

      $W_L < W_R$: the conversion preserves value.

      $W_L = W_R$: (two sub-cases)

         $S_L = S_R$: the conversion preserves value.

         $S_L != S_R$: the conversion is bit-preserving: the bit

pattern of V_R is stored into V_L. Value is preserved if and only if V_R is a value in R.

b. R is a floating type and L an integral type.

If V_R is non-negative, V_R' is the largest integer not greater than V_R; if V_R is negative, whether V_R' is the largest integer not greater than V_R or the smallest integer not less than V_R is implementation-defined.

c. R is an integral type and L a floating type.

V_R' is the floating value in L nearest V_R. If there is no unique nearest value, the result is implementation-defined. If V_R is outside the range of values of L, the result is undefined.

d. R is Float and L Double.

$V_R' = V_R$.

e. R is Double and L Float.

V_R' is V_R rounded to the nearest value in R. If there is n unique nearest value, the result is implementation-defined. If V_R is outside of the range of values representable in L, the result is undefined.

Examples: Assume that **long ints** are wider than **ints** and that **double** has more range and precision than **float**.

```
int *p1,*p2;
struct {int x;} st1;
struct {int x;} st2, st3;
int i; long int l; float f; double d; unsigned u;
p1 = p2;      /* Legal.                                          */
st1 = st2;    /* Illegal.                                        */
st2 = st3;    /* Legal.                                          */
i = l;        /* Conversion truncates.                           */
l = i;        /* Conversion preserves value.                     */
u = i;        /* Conversion preserves bits; may preserve value.  */
f = i;        /* Value nearest i in Float is stored in f.        */
d = f;        /* Exact value stored.                             */
f = d;        /* Value nearest d in Float is stored if it exists.*/
```

3.14 Integral Widening Conversions

[arithmetic conversions; unsignedness-preserving versus value-preserving]

Many contexts require type Signed-Int or Unsigned-Int and "widen" any shorter integer types to one of these types. The widening is defined as follows:

```
Widen(T) =
if    T is Signed-Char or Signed-Short-Int
then  Signed-Int
else  if    T is Unsigned-Char or Unsigned-Short-Int
      then  if the size of T is the same as that of
                Unsigned-Int on the target architecture
            then  Unsigned-Int
            else  Signed-Int
      else  undefined.
```

4.2BSD provides slightly different widening: unsigned types always widen to Unsigned-Int. This rule, often called "unsignedness-preserving" as opposed to the "value-preserving" Widen defined above, produces quite a few surprises, as illustrated by the following C text:

```
void f() {
   unsigned char c = getchar();
   if (c - '0' < 0 || c - '0' > 9)
      printf("Non-digit. \n");
   }
```

In 4.2BSD "c - '0'" is never negative, since it is an unsigned type (see the next Subsection for combination of operands in '-'), so the first test always fails. In KR, there is no Unsigned-Char or Unsigned-Short-Int type so Widen agrees with KR. X3J11 also uses the widening rules of Widen.

3.15 Combination of Operand Types

[arithmetic conversions]

Binary operators may operate on values of different types When this happens, generally both values are converted to a common type, which is the result type of the operation. The function Common(T_1,T_2) specifies the common type T of two operand types T_1 and T_2 that are both arithmetic types. It is defined as follows:

1. An implementation may employ either rule (a) or (b):
   a. If either T_1 or T_2 is Long-Double, T is Long-Double; otherwise, if either T_1 or T_2 is Double, T is Double; otherwise, if either is Float, T is Float.
   b. If either T_1 or T_2 is Long-Double, T is Long-Double; otherwise, if either T_1 or T_2 is Float or Double, T is Double.

2. Or, if one is Unsigned-Long-Int, T is Unsigned-Long-Int;
   or, if one is Signed-Long-Int, T is Signed-Long-Int;
   or, if one is Unsigned-Int, T is Unsigned-Int;
   or, if one is Signed-Int, T is Signed-Int;
   otherwise, T is Common(Widen(T1), Widen(T2))
   (widenings guarantee that one of the above cases applies).

3.16 Expression Evaluation: Side Effects, Sequence Points

[comma operator]

The semantics of an expression is two-fold, involving the production of a value, called the *evaluation* of the expression, and potential "side effects".

A *side effect* is an aspect of an expression's semantics that need not occur during the evaluation. However, If an expression is evaluated, any associated side effects must have taken place by certain points called *sequence points.* Thus, side effects may occur any time from the inception of the evaluation of the expression up to the next sequence point. The term and concept of a "sequence point" are borrowed from X3J11.

The end of execution of any statement or declaration is a sequence point. This means that by the end of the execution of the statement or declaration, all pending side effects must have occurred.

Sometimes the end of the evaluation of an expression is a sequence point. This means that by the end of the evaluation of the expression, all pending side effects must have occurred.

In Section *Expressions* we specifically note which expressions have side effects, and in any section where an expression is used we specifically note if its evaluation is a sequence point. In most cases the evaluation of an expression contained within a statement is a sequence point.

Example: There are two side effects in the expression "`i++ + j++`": the first increments the variable `i`, and the second the variable `j`. However, the side effects need not occur during the evaluation of the expression or during the evaluation of the subexpressions `i++` and `j++`. But after the execution of the statement "`k = i++ + j++;`", the side effects must have occurred, so a use in a subsequent statement of `i` or of `j` must access the incremented value.

In "`k = i++ + j++;`", there is no question what k will be assigned, assuming the values of `i` and `j` are known.

Example: But consider the more interesting case:

```
int a, b, c, k, i = 0;
k = (a = i++) + (b = i++) + (c = i++);
```

An implementation is free to evaluate the three assignment expressions in any order (see Section *Expressions*); assume for the moment it is left-to-right. What one might expect is that a is assigned 0, b 1, and c 2. But since the incrementation side effects can be delayed to the end of the expressions, all of a, b, and c can be assigned 0. Thus k can take on any of the values 0, 1, 2, or 3, depending upon when side effects occur.

Example: Introducing sequence points with a comma operator can eliminate uncertainty. Consider:

```
int a, b, c, k, i = 0;
k = (a = (i++, i-1))+(b = (i++, i-1))+(c = (i++, i-1));
```

Here, independent of the evaluation order of the assignment subexpressions, the values assigned will be 0, 1, and 2, so k will obtain the value 3. But it is still not known *which* of a, b, and c obtain the value 0, 1, and 2, since the order of evaluation of the operands of + is unspecified.

4
Lexicon

4.1 Character Set

[target and host character set; string terminator]

Two sets of characters are implementation-defined: that interpreted by the target machine (the *target* set), and the *host* set in which C program text is written. The distinction matters for the string and character words that are written with the host set and translated into the target set.

A character whose representation has all bits zero must exist in the target character set, representing the character used to terminate strings.

The host character set must contain the following characters: the space character; characters representing audible alert, backspace, horizontal tab, new line, vertical tab, form feed, and carriage return; the 52 uppercase and lowercase characters of the English alphabet; the 10 decimal digits; and the following 29 graphic characters:

```
! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { | } ~
```

4.2 Line Splicing

[end-of-line; "\"]

C program text may be divided into a sequence of lines. Each line is terminated by what is designated in the grammar below as an "Eol", for end-of-line. The host environment may terminate lines with a special character, by keeping track of the record length of each line, or by some other technique.

No matter how this is done, each line that ends with the character "\" is considered to be "spliced" with the next line, as if the line terminator were not present. This permits any C word to be longer than a single line. For example:

This	*is equivalent to*
`char *s = "a long\` `string";`	`char *s = "a longstring";`

`short_id = a_longer_id\`
`entif\`
`ier;`
	`short_id = a_longer_identifier;`

`/* This is a \` `comment. */`	`/* This is a comment. */`
`X = \` `3;`	`X = 3;`

Line splicing is not formally treated in the lexical grammar, even though it is possible, because it would increase the grammar's complexity out of proportion to its importance.

Discussion

Line splicing is not a part of KR or 4.2BSD. There, \ is used only for continuing long strings to the next line.

X3J11 decided to generalize the construct so that it could always be used to overcome line length limitations on some operating systems.

Although it will rarely be used by programmers for anything other than continuing strings, it may simplify program-generating programs.

4.3 Preprocessor and Lexicon

[line boundaries]

Part of the definition of C includes a so-called *preprocessor* that was introduced in Section *Notation* as effecting conditional compilation, file inclusion, and macro substitution.

The conditional compilation aspect of the preprocessor necessitates two distinct lexical descriptions of C source text: that text excluded from compilation, and that text included in compilation. These two descriptions are contained in the single lexical grammar presented here.

The preprocessor is the only part of C where line boundaries play a significant role. Commands to the preprocessor are terminated by line boundaries, and conditionally compiled text consists only of complete lines, not partial lines.

The preprocessor lexical conventions do not fit well with the rest of the C language, so a substantial part of the lexical grammar is devoted to the preprocessor. Much of this part, and the semantics of the preprocessor — i.e. the rules and effects of conditional compilation, file inclusion, and macro substitution — are described in Section *Preprocessor.*

4.4 Included and Excluded Text

Syntax

scanner C_lexicon:

```
C_lexicon -> Text;
  Text -> (Words Line_end)* (Control_line Text)?
       -> \Scanning Skipped_lines Control_line Text;
```

An overall C program, on the lexical or word level, is one of two forms of Text: Text that is included in compilation (the first alternative), or Text that is excluded from compilation.

Included Text consists of a sequence of lines each of which contains Words. This sequence of lines is terminated by either the end of the program or a Control_line (preprocessor directive) followed by more Text.

Excluded Text consists of a sequence of Skipped_lines. That is followed by a Control_line, then more Text. Control_line is needed after Skipped_lines so that it will be included rather than skipped.

Whether to include or exclude text is determined by the preprocessor and is described in the grammar by "\Scanning". "\Scanning" in the second rule for Text means to use that rule if *not* scanning, i.e. if skipping.

The complexity of the grammar at this level is due entirely to the line orientation of the preprocessor.

4.5 Words

Syntax

```
Words  -> Word*;
Word   -> String | Char | Number | Identifier
        | Delimiter | Punctuator | Operator | Comment;
```

The Words of C are Strings, Chars, Numbers, Identifiers, etc.

4.6 Identifiers

Syntax

```
Identifier-> Id_text                          =>'<IDENTIFIER>';
  Id_text -> Letter (Letter | Digit)*;
  Letter  -> 'A'..'Z' | 'a'..'z' | '_' ;
```

The nonterminal Id_text is used also in describing Control_ lines and hence cannot be back-substituted.

Constraints

Each character in an Identifier stands for itself, and distinction is made between upper and lower case. Thus the Identifiers "abc" and "ABC" are regarded as different: two Identifiers are the same only if they consist of exactly the same sequence of characters. There is no constraint on the length of an Identifier.

Semantics

The text of an Identifier has no semantics at all. Identifiers serve only to relate declarations of things with their subsequent uses; an executing C program deals only with the declared things and has no need for Identifier texts.

Discussion

KR specifies that only the first eight characters of each Identifier are significant. In addition, linkers on some machines accept names of limited length. 4.2BSD permits names of unlimited length. X3J11 requires an implementation to treat as significant the first 31 characters of each name not having static-export or static-import storage class.

4.7 Numbers

[<INTEGER>, <OCTAL>, <HEX>, <FLOAT>; Signed-Int, Unsigned-Int, Signed-Long-Int, Unsigned-Long-Int]

Syntax

```
Number    -> Integer | Octal | Float | Hex ;
Integer   -> '1'..'9' ('_'? Digits)? Integral_suffix?
                                          =>'<INTEGER>';
Octal     -> '0' ('_'? Oigits)? Integral_suffix?
                                          =>'<OCTAL>';
Hex       -> '0' ('X' | 'x') Higits Integral_suffix?
                                          =>'<HEX>';
Float     -> Mantissa Exponent? Float_suffix?=>'<FLOAT>'
          -> Digits   Exponent  Float_suffix?=>'<FLOAT>';

  Mantissa-> '.' Digits | Digits \Dot_dot '.' Digits?;
  scanner Dot_dot: Dot_dot -> '.' '.'; end Dot_dot
  Exponent-> ('E'|'e') ('+'|'-')? Digits;

  Integral_suffix
          -> 'u' 'l'? | 'l' 'u'? | 'U' 'L'? | 'L' 'U'?;
  Float_suffix
          -> 'L' | 'l' | 'F' | 'f';

  Higits  -> Higit+ list '_';          # _ is non-standard.
  Higit   -> '0'..'9' | 'A'..'F' | 'a'..'f' ;
  Digits  -> Digit+ list '_';          # _ is non-standard.
  Digit   -> '0'..'9' ;
  Oigits  -> Oigit+ list '_';          # _ is non-standard.
  Oigit   -> '0'..'7' ;
```

The notation "\Dot_dot" indicates that Digits followed by '.' is a Mantissa only when what follows the Digits is *not* a Dot_dot, which is a scanner for '.' '.'. This disambiguates the construction "**case** 1..2:" permitted in a **switch** statement.

Four forms are described: Integer, Octal, Float, and Hex.

As mentioned in Section *Notation*, the longest possible interpretation prevails: therefore "12" denotes the Number 12 instead of the Numbers "1" and "2".

Constraints

The type of an <INTEGER> is Signed-Int unless its value is not a Signed-Int value, in which case it is Signed-Long-Int, but with the following exceptions: if the Integral_suffix 'u' or 'U' is employed, its type is the unsigned variety of the type just determined; and if 'l' or 'L' is employed, the type is the long variety of the type just determined. Thus "123ul" is of type Unsigned-Long-Int.

If a <HEX> or <OCTAL> constant has a Signed-Int value, its type is Signed-Int; otherwise, if it has an Unsigned-Int value, its type is Unsigned-Int; otherwise, if it has a Signed-Long-Int value, its type is Signed-Long-Int; otherwise its type is Unsigned-Long-Int. Exception: the use of the Integral_suffix modifies the type just determined in the same way as for <INTEGER>, explained in the prior paragraph.

The "value" of a constant is defined in *Semantics* below.

The type of a <FLOAT> is Double, unless it is suffixed by 'l' or 'L', in which case it is Long-Double, or by 'f' or 'F', in which case it is Float.

Example: 0xffff on a machine with two-byte Unsigned-Ints and four-byte Unsigned-Long-Ints is the Unsigned-Int value 65_535; 0x10000 is the Signed-Long-Int value 65_536; 0x7fff_ffff is the Signed-Long-Int value 2_147_483_647 and 0x8000_0000 is the Unsigned-Long-Int value 2_147_483_648.

Semantics

Underscores have no significance in Numbers.

Integers are interpreted in base 10.

Octal numbers are interpreted in base 8.

Floats are interpreted in the standard fashion. The letter 'e' or 'E', if present, means "times ten-to-the-power-of" the (optionally signed) base-10 integer after the 'e' or 'E'.

The characters following the 'x' or 'X' in Hex are interpreted as being in base 16. The letters 'A' through 'F' (or 'a' through 'f') denote the values 10_{10} through 15_{10}.

Discussion

Underscores in Numbers are an extension (from Ada) over all C definitions and implementations known to us. They are allowed so that long numbers can be broken up for ease of reading. They may be used as a replacement for comma to separate thousands: 3_434_112, for example. Following Ada, High C disallows such forms as 1_____ to denote the integer 1: each underscore must appear between two digits.

In KR and 4.2BSD, the digits 8 and 9 are permitted in an octal constant and have values 10_8 and 11_8, respectively. Both X3J11 and High C disallow such.

The Integral_suffix and Float_suffix appear only in X3J11 and here. 'f' or 'F' can be used to prevent arithmetic operations from being performed in Double precision in the presence of a constant; see Section *Expressions*. But note that it cannot be used to pass a Float constant as a Float parameter instead of a Double in the absence of a function prototype, since in "g(1.0f);", if there is no prototype for g, 1.0 is passed as Double. See later discussions on parameter passing and function definition.

At one time the ".." word was allowed by X3J11 for specifying ranges in a **case** statement. It was discarded because the committee felt that the task of preventing "1..2" from

being recognized as two Floats (expressed by the \Dot_dot in the grammar) was distasteful. There is no inherent ambiguity in the language definition, since on the phrase-structure level it is not possible for two Floats to be adjacent. Therefore "1..2" must *always* be "1", "..", and "2" in a legal program.

Machine dependencies

The range and precision of real numbers are machine dependent.

4.8 Strings and Characters

[<STRING>, <CHAR>; octal, hexadecimal in strings and characters; Signed-Int; "\"; ASCII; arrays]

Syntax

```
String    -> '"'  DQchar* '"'                    =>'<STRING>';
Char      -> ''''  SQchar ''''                   =>'<CHAR>';
  DQchar  -> Any-'\'-'"'  | '\' Special ;
  SQchar  -> Any-'\'-''''  | '\' Special ;
  Special -> 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v'
           | '\' | '"' | ''''
          -> Oigit (Oigit Oigit?)?               # Octal.
          -> 'x' Higit (Higit Higit?)?;          # Hexadecimal.
```

We describe two words here: the character Char and string String. In both, each character in the target character set is represented by a character in the host character set or by an octal or hexadecimal escape sequence.

Any character the user can type at the terminal may appear within a String or as a Char.

Constraints

Each <CHAR> has type Signed-Int. Each <STRING> has type *T, where T is the type of c in "**char** c;", except when appearing as an Initializer of an array or as an argument to **sizeof**, when it has type [V]:T, where V is the number of characters in the string; see Sections *Expressions/**sizeof*** and *Declarations/Non-Function Definitions.*

Semantics

Char. The enclosing apostrophes for a Char have no meaning themselves, but merely delimit the enclosed character. The value of a character is the numerical value of the quoted character in the target machine's character set.

Note that backslash (\) is not permitted as a single enclosed character. When \ appears after the first apostrophe, the \ itself and the character following it together have one of the meanings indicated in the table below. (The ASCII value column is applicable only when the target character set is ASCII, and is provided here for convenience.)

Pair	*Meaning*	*ASCII value (in decimal)*
\a	audible alert	7
\b	backspace	8
\f	form feed	12
\n	new line	10
\r	carriage return	13
\t	horizontal tab	9
\v	vertical tab	11
\\	\	
\'	'	
\"	"	

\d binary value corresponding to octal digit d.
\dd binary value corresponding to octal digits dd.
\ddd binary value corresponding to octal digits ddd.
Constraint: In these last three cases the binary value must fall into the range allowed for characters on the target machine. Each "d" must be an Oigit as defined in the grammar.

\xd binary value corresponding to hexadecimal digit d.
\xdd binary value corresponding to hexadecimal digits dd.
\xddd binary value corresponding to hexadecimal digits ddd.
Constraint: In these last three cases the binary value must fall into the range allowed for characters on the target machine. Each "d" must be a Higit as defined in the grammar.

In the forms \ddd and \xddd, once again the "longest text" rule of Section *Notation* applies. For example, '\78' is an illegal Char: it would be two characters, \7 and 8.

String. The enclosing quotes for Strings have no meaning themselves, but are merely delimiters of the string text. Each enclosed character denotes its corresponding character in the target character set, with the exception of \ and ". When \ appears, it and the character following it together have the meaning as indicated in the above table for Char.

The value of a String S depends upon its type T; see *Constraints* above. If T is a pointer type, the value is a pointer to an array A of characters such that A[i] is the i^{th} enclosed character, for $0 \le i < L$, and A[L] is the value of the character '\000'. Here L is the number of characters in S. If T is an array type, the value of S is A.

Discussion

The Char 'X' does *not* have the same value as the String "X". Confusing these is a common mistake in C and can cause disastrous results. For example, if function G expects a character string, the invocation "G('x');" may cause G to run through arbitrary amounts of memory looking for the terminating '\000'.

\a, \v, and hexadecimal escape sequences are inventions of X3J11 not in KR.

KR and 4.2BSD permit the "default" case of \C, where C is not in the table above; in this case \C means the same as C.

Examples:

Characters:	*Strings:*
'A'	"A"
'\''	"He said, \"She's dead.\""
'\t'	"THIS IS A STRING."
';'	"Praise God from whom all blessings flow."

4.9 Operators

Syntax

```
Operator  -> AssignOp | OtherOp;
  OtherOp -> '~' | '&' '&' | '|' '|' | '>' '='
           | '<' | '=' '=' | '!' '=' | '<' '='
           | '>' | '+' '+' | '-' '-' | '-' '>'
           | '!' | '='     | '.'     | '?'   =>'<AS_IS>';

  # Operators that can be followed by '=' in assignments.
  AssignOp->('^' | '>' '>' | '<' '<'
            |'+' | '*'     | '&'     | '%'
            |'-' | '/'     | '|'
            ) '='?                          =>'<AS_IS>';
```

Semantics

Operator meanings are described in Section *Expressions.*

Discussion

KR and 4.2BSD permit the anachronism "x =+ 1" instead of the recommended "x += 1". The former can be misinterpreted as assigning the value +1 to x rather than incrementing x. X3J11 forbids the anachronism, as do we.

4.10 Punctuators

Syntax

```
Punctuator-> '(' | ')' | ';' | '.' '.'
           | '[' | ']' | ',' | '.' '.' '.'
           | '{' | '}' | ':'               =>'<AS_IS>';
```

Semantics

Punctuators (or "punctuation marks") have no semantics in and of themselves. They separate other words in C, and their placement is constrained by the phrase-structure syntax.

Discussion

The punctuator ... was added by X3J11 to allow specifically designating a function as callable with a varying number of arguments. .. is a High C extension to support ranges in the **switch** construct. Neither punctuator is in KR or 4.2BSD.

4.11 Delimiters and Eol

[white space]

Syntax

```
Delimiter -> ( Space:    ' '+   | HorizTab: 'ht'
             | FormFeed: 'ff'   | Vert: Tab: 'vt'
             )+                                  =>'<DELETE>';

Line_end  -> Eol                                 =>'<DELETE>';
```

Note that Delimiters are not involved in the phrase-structure definition of C programs, since Delimiters are words named <DELETE>.

Due to the line orientation of the preprocessor, Eol is not a Delimiter as might be expected.

Semantics

Delimiters, commonly called "white space", are entirely insignificant. Their only purpose is to separate words. Eol separates sequences of included or excluded Text and terminates Control_lines (see below).

Discussion

The addition of the FormFeed character is an extension over KR. In addition, KR dictates that Eol is the new-line character. This is not necessary on systems that support a different convention for line termination.

4.12 Comments

Syntax

```
Comment  -> '/' '*' Rest                          =>'<DELETE>';
  Rest   -> Most* '*'+ ('/' | (Most-'/') Rest) ;
    Most -> Any-'*' | Eol;
```

Semantics

Comments have no effect on the program's meaning.

Discussion

The grammar for Comment is a precise description of comments that begin with the two adjacent characters "/*" and end with the two adjacent characters "*/". "Adjacent" implies that an Eol cannot appear between the two characters, even if the Eol is not really a character but a file-system end-of-record, for example.

Comments do not nest. Commenting out code containing Comments can be achieved by using the C preprocessor (see Section *Preprocessor*), as follows:

```
#if 0
... commented out code ...
#endif
```

A good way to write large block comments with standard Comments is exemplified below:

```
/*
 *  This is a
 *  block comment
 *  whose left edge is
 *  lined with *s
 *  to make it stand out.
 */
```

4.13 Excluded Text

Syntax

```
Skipped_lines   -> (\Sharp Skipped_line)*;
scanner Sharp: Sharp -> '#'; end Sharp
Skipped_line    -> Skip_suffix? Line_end;
Skip_suffix     -> ( Not_special | Slash
                   | Comment | DString )*;
Not_special     -> (Any - '/'-'"')+          => '<DELETE>';
Slash           -> '/'                       => '<DELETE>';
DString         -> String_text               => '<DELETE>';
```

This is the description of lines that are excluded from compilation. Control_lines are the only lines that are not skipped, and even then only Control_lines not within Comments. The notation "\Sharp" means that each Skipped_line must *not* begin with a Sharp ('#'), thereby ensuring that Control_lines are not skipped.

Skipped lines conform to a small amount of lexical syntax: Comments and Strings must be well-formed. Comment syntax is included is to permit Control_lines to be part of Comments, i.e. to allow Control_lines within excluded Text to be "commented out". String syntax is included so that "/*" in a string is not misinterpreted as the beginning of a Comment.

4.14 Control Lines: Preprocessor Commands

Due to the complexity of the preprocessor syntax, both its syntax and semantics are deferred to the next section, *Preprocessor*. Thus the lexical grammar in this section is, as it stands, incomplete.

4.15 Reserved Words

Syntax

```
end      C_lexicon
reserve '<IDENTIFIER>'
```

The following are words appearing in the phrase-structure grammar (discussed in the next section). As discussed in Section *Notation* each word below is therefore not an <IDENTIFIER> word, but instead a reserved word. The implication is that the C programmer may not use these words as user-coined <IDENTIFIER>s:

int char float double long short unsigned signed struct union typedef auto static extern register goto return break continue if else for do while switch case default entry sizeof void enum pragma

Discussion

Some implementations also reserve the words **asm** and **fortran**. **signed** is not reserved in 4.2BSD or KR, but is in X3J11. **pragma** is reserved only in High C.

5

Preprocessor

5.1 Introduction

[conditional compilation; file inclusion; macro replacement; lexicon; phrase-structure; preprocessor commands; lexical analysis; feedback]

As mentioned in the previous section, the C preprocessor does three things: conditional compilation, file inclusion, and macro replacement. In this section we define what these terms mean.

The preprocessor may be thought of as a level of description and program source transformation that lies roughly between the lexicon and the phrase structure of C. However, this is not a precise division since the preprocessor commands themselves are embedded in C program source text, must therefore be lexically described, and hence impact the lexicon of C. Furthermore, due to the semantics of file inclusion, lexical analysis and preprocessing cannot be separated. This is described in greater detail in file inclusion, below.

Another way of looking at the preprocessor is as a filter that takes the word sequence WS generated by lexical analysis and produces a new sequence of words subject to C phrase-structure analysis. Some of the words in WS are interpreted as preprocessor commands, and are processed and discarded by the preprocessor; others are replaced by other words; the rest of the words are left alone. In a program with no preprocessor commands, the preprocessor does nothing to WS, and all of WS is subject to the C language phrase-structure analysis.

However, again due to the requirement that lexical analysis and preprocessing must occur at the same time, the filtering cannot be separated into a distinct pass but must occur during lexical analysis. (When the preprocessor is implemented as a separate pass, it is at the expense of redundantly doing most lexical analysis in both the preprocessor and compiler *per se*.)

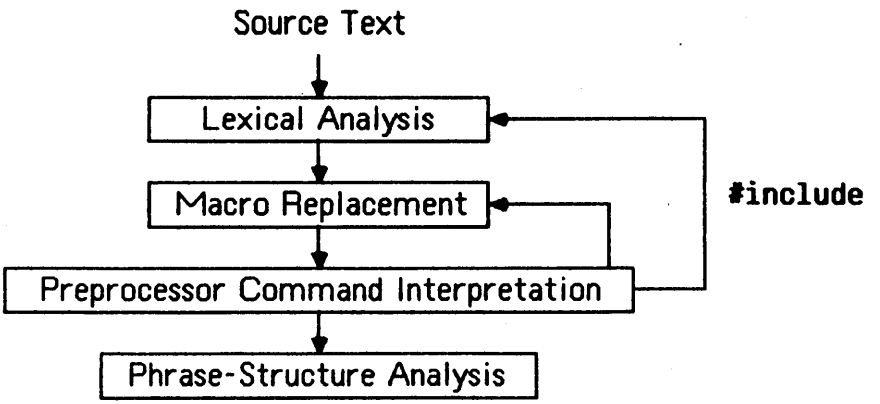
The preprocessor "filter" is broken up into two pieces: the macro replacement (MR) phase, and the command interpretation (CI) phase. MR occurs immediately after lexical analysis. CI occurs after MR, which means that some of the preprocessor commands may have been subjected to macro replacement. *Example:*

```
#define RELEASE 2
#if RELEASE > 1
... some text T
#endif
```

When CI processes the "**#if**" command, the text "RELEASE" has already been replaced with "2", so that CI includes text T (because 2 > 1).

However, some preprocessor commands must avoid macro replacement. For example, "**#ifdef** RELEASE", which asks if RELEASE is defined, will not work if RELEASE is replaced with "2", since CI will instead process "**#ifdef** 2". Therefore CI must occasionally instruct MR to avoid replacement.

Finally, due to the semantics of file inclusion, CI occasionally directs the lexical analysis phase to include text from a file. Pictorially, the relationship between MR, CI, and other phases of a C language translator is as follows:



The arrows leading back from CI to MR and Lexical Analysis indicate the occasional "feedback" from CI to the other two

phases. This feedback is the single reason that preprocessing and lexical analysis cannot be separated into separate passes.

In this section we first present the portion of the C lexical grammar that describes the preprocessor commands and that was deferred from Section *Lexicon.* This completes the lexical description of C.

Then, the phrase structure of the preprocessor is presented — how the preprocessor interprets the words determined by lexical analysis, including the ferreting out of preprocessor commands. The semantics of the preprocessor commands are described relative to the preprocessor phrase structure.

5.2 Control Lines

Syntax

```
Control_line -> Sharp Delimiter? Control;
Sharp        -> '#'                          =>'<DELETE>';
```

Observe that the Sharp of a Control_line lies in column one of an input line due to the placement of Control_line in the productions for Text (see the previous section), where it appears either: (a) at the beginning of the source input; (b) after a Line_end; or (c) after Skipped_lines, which always end in a Line_end.

5.3 "Comment" Control Line Lexicon

Syntax

```
Control -> Line_end;
```

If nothing appears after the # and optional trailing Delimiter on a control line, the text is all deleted and the line is effectively a comment.

5.4 Macro Definition Lexicon

Syntax

```
Control -> /Define_word Define Delimiter Macro
           Words Done Line_end
 scanner  Define_word:
          Define_word  -> 'd' 'e' 'f' 'i' 'n' 'e';
   end    Define_word
 Define -> Define_word                    =>'<CONTROL>';

 Macro  -> Id_text \LP                    =>'<NO_PARMS>'
        -> Id_text /LP                    =>'<WITH_PARMS>';
 scanner  LP: LP  -> '(' ;  end LP
```

Macro definition requires special lexical processing that complicates the grammar — yet another place where C preprocessor syntax is not well-designed.

The placement of a left parenthesis as the next character following a macro name (Id_text) has different semantics from a left parenthesis separated from the macro name by one or more characters. The difference is conveyed by the distinct word names (<NO_PARMS> versus <WITH_PARMS>) for the Id_text, and is further explained below in Section 5.8.

The /Define_word resolves an ambiguity among Control_ lines, since macro definition is lexically a subset of other control lines. If **define** appears immediately after the Sharp and optional Delimiter, this rule for Control takes precedence over all other alternatives for Control.

5.5 Other Control Line Lexicon

[else in lexical grammars]

Syntax

```
Control  -> (Include_text else Other_control)
            Words Done Line_end;

Done     ->                                    =>'<C-EOL>' ;

Other_control  -> Id_text                      =>'<CONTROL>';

scanner Include_text:
  Include_text  -> Include_word Delimiter? Funny_string;
  Include_word  -> 'i' 'n' 'c' 'l' 'u' 'd' 'e'
                                               =>'<CONTROL>';
  Funny_string  -> L_angle File_name R_angle;
  L_angle       -> '<'                         =>'<DELETE>';
  R_angle       -> '>'                         =>'<DELETE>';
  File_name     -> (Any-'\'-'"'-'>'  | '\' Special)*
                                               =>'<<STRING>>';
  end Include_text
```

The grammar-reserved word **else** serves to resolve an ambiguity, since Include_text and Other_control can generate some identical character sequences. **else** used here forces the Include_text interpretation to prevail where a conflict exists. This rule for Control may not be used when the macro definition rule can be used; i.e. the macro rule takes precedence.

All preprocessor commands end at line boundaries. The word <C-EOL> marks those boundaries and allows the preprocessor semantic phase to distinguish preprocessor commands from other C text. Note that Done precedes Line_end everywhere it appears and therefore always marks a line boundary.

Include_text describes one of the sloppiest parts of the C preprocessor syntax: the form "**#include** <string>" of file inclusion. Here a string is delimited by < and > rather than the standard double-quotes (") provided elsewhere in the C lexicon. The cost is a duplication of some of the string definition syntax. A better language design would have been to use a word

other than "**#include**": "**#L_include** "string"" for "library include", for example.

5.6 Control Line Phrase Structure

Now we present the phrase structure of the preprocessor. The grammar describes completely the sequence of words determined by lexical analysis, but recall that the preprocessing is interleaved with the lexical analysis, so that the grammar cannot be understood to be a description of an already-generated sequence of words: the semantics of preprocessing changes that sequence of words.

Note that the preprocessor "comment" discussed in Section 5.3 above need not and is not addressed in the phrase structure.

Some new notation: `'wordname'!'text'` in the grammar below indicates the word whose name is `'wordname'` and whose text is `'text'`. The preprocessor phrase-structure grammar is the only grammar that depends upon the text of some words.

Syntax

```
Text -> (Control_text | Word)*;
```

From the preprocessor's point of view, Text has little structure: it is just a sequence of Words, occasionally interrupted by preprocessor commands and the text enclosed by such commands (Control_text).

5.7 File Inclusion

[include]

Syntax

```
Control_text
-> '<CONTROL>'!'include'   '<<STRING>>' '<C-EOL>'
-> '<CONTROL>'!'include'    '<STRING>'  '<C-EOL>'
-> '<CONTROL>'!'c_include' '<STRING>'  '<C-EOL>';
```

Semantics

A file inclusion preprocessor command substitutes an entire source file F for the command. The substitution must occur immediately after processing the command and during lexical analysis, so that the the contents of F may also be subjected to lexical analysis. Thus the concept of preprocessor as filter concept is accurate only if the preprocessing is thought of as proceeding concurrently with the lexical analyzer.

Due to the semantics of file inclusion this concurrency can be avoided only if the preprocessor is run as a separate pass *preceding* lexical analysis, in which case the preprocessor must have its own lexical analyzer within it — but this just shows again that lexical analysis and preprocessing must happen at the same time.

A substituted source file is considered to be terminated by a Line_end even if it is not. This prevents a word starting at the tail end of a substituted source file from being "continued" in the file containing the **#include** command.

The first two **#include** commands differ only in how the preprocessor locates the file; this is implementation-defined. The last command ("conditional" include) differs from the second only in that file inclusion does not occur if the file has already been included.

In all three forms the name of the file searched for is the text of the <<STRING>> or <STRING> word.

Discussion

The conditional include concept appears only in High C.

5.8 Macros

[macro parameters and arguments; macro body; replacement text; macro replacement; **#define** and **#undef**; parameterless and parameterized macros; benign redefinition]

C includes the notion of a *macro* facility. A macro is an identifier M associated with some *macro body* and possibly some *macro parameters.*

An occurrence of M causes it and possibly other words following it to be replaced by a different sequence of words called the *replacement text* which is derived from the macro body and possibly some supplied *macro arguments.* This is called *macro replacement;* it is more precisely defined later.

Macro replacement is performed on the word sequence produced by lexical analysis before the preprocessor analyzes that sequence for preprocessor commands, i.e. before any analysis based on the preprocessor phrase-structure grammar (PPSG) is performed. The nonterminal Words in the PPSG represents Words upon which all applicable macro replacement has been performed, with exceptions as noted. The semantics of macro replacement is defined formally below.

Syntax

```
Control_text
-> '<CONTROL>'!'define'
   ( Mname:'<NO_PARMS>' Body:Words
   | Mname:'<WITH_PARMS>'
     '(' Parm:'<IDENTIFIER>' list ','? ')' Body:Words
   )
    '<C-EOL>'
-> '<CONTROL>'!'undef' Mname:'<IDENTIFIER>' '<C-EOL>'
;
```

Constraints

In the **#define** command, if the Mname is already defined as a macro M, its definition must be the same as M, where same means the parameter <IDENTIFIER>s and the Body must be identical. (This is known as "benign redefinition".)

No two Parm:<IDENTIFIER>s in the <WITH_PARMS> form of the **#define** may be the same, i.e. have the same text. Stated another way, macro parameter names must be distinct from each other.

Semantics

Both forms of the **#define** preprocessor command define an identifier to be a macro with body Body:Words. No macro replacement is performed on the Body of a **#define** command.

In the first form of **#define**, that with the <NO_PARMS> macro name, the macro is called a *parameterless* macro, and subsequent instances of the macro are replaced by the Body:Words. Note that <IDENTIFIER> is a word with text; the text is the macro's name.

In the second form of define, that with the <WITH_PARMS> macro name, the macro has a list of zero or more parameter <IDENTIFIER>s, and is called a *parameterized* macro.

For macro replacement to occur for a parameterized macro M, the occurrence of M must be followed by a left parenthesis, a number of arguments that match the number of parameters, and a concluding right parenthesis. The structure of the arguments is defined later. Supplying the wrong number of arguments is an error.

The replacement text that replaces M and its parenthesized argument list is the macro body with any occurrence of a parameter <IDENTIFIER> in the body replaced by the corresponding actual argument from the argument list.

In all cases, after macro replacement has occurred, the replacement text is reprocessed for any other possible replacement, with the exception that macro replacement does not occur on a macro named M contained directly or indirectly in text resulting from a replacement of M. The exception prevents an endless loop in macro expansion. Furthermore, macro replacement occurs on arguments to a macro M before M itself is subjected to replacement.

We more formally specify macro replacement below by defining $f_F(WS)$, the result of macro replacement on word sequence WS with respect to set F of macro names whose definitions must temporarily be "forgotten" so that endless loops can be avoided. One might read f_F as "forget F while substituting in WS". $f_{\{\}}(WS)$ is the result of macro replacement on the word sequence WS resulting from lexical analysis. In the definition || denotes concatenation of word sequences, and W is a single word.

$f_F(WS) =$

if WS is the empty sequence,
then WS;

else if WS is of the form W || WS', **and** W is not a macro name **or** W is an element of F,
then W || f_F(WS');

else if WS is of the form W || WS' **and** W is a macro without parameters,
then $f_{F \cup \{W\}}$(Rep(W)) || f_F(WS');

else if WS is of the form W || '(' || Arguments || ')' || WS', **and** W is a parameterized macro with n formal parameters, **and** there are n Arguments (defined below) $A_1, A_2, \ldots, A_n$,
then $f_{F \cup \{W\}}$(Rep(W,A_1',A_2',..., A_n ')) || f_F(WS'),
where $A_i' = f_F(A_i)$;

else WS is of the form W || WS' where W is a parameterized macro lacking the appopriate number of parameters, and the result is W || f_F(WS').

Rep(W,A_1,A_2,...A_n) is the replacement text for macro W with n arguments A_1,A_2,...A_n, derived from W's macro body B by replacing each occurrence of the i^{th} parameter <IDENTIFIER> in B with A_i.

The Arguments to a parameterized macro are described by the following subgrammar:

```
Arguments  -> '(' W* list ',' ')';
W          -> Word - '(' - ',' - ')'
           -> '(' (W | ',')* ')';
```

After a **#undef** command, the named macro Mname is no longer a macro (its definition is "forgotten"). If Mname is not a macro, the **#undef** command has no effect.

Discussion

Essentially an argument to a parameterized macro may not contain "," except within properly balanced parentheses. But note that an argument can be an empty sequence of words, so that in

```
#define call(f,arg) f(arg)
...
call(,3)
```

"call(,3)" is replaced by "(3)".

In addition, since an occurrence of a parameterized macro is not replaced unless the appropriate number of arguments is present, one can ensure that a library function being invoked is not a macro by parenthesizing the function:

```
#include "stdio.h"
...
   /* If getc is a macro, it is replaced here:      */
c = getc(F);
   /* But here the macro is not replaced;           */
   /* instead, function getc is called:             */
c = (getc)(F);
```

Benign redefinition has been adopted by X3J11 as a way to permit the inclusion of a commonly used macro name, such as "NULL" (typically defined as "**(void** *)0"), in each library header file that requires it.

Bug. Our definition of macro replacement avoids some of the absurdities and possible abuses of common C macro preprocessors. In at least two C compilers we know of, a macro invocation can be constructed from text some of which was obtained from a macro replacement and the rest elsewhere.

For example, the following is acceptable to these two compilers, producing "**int** i = (3)*(3);" from the expansion:

```
#define start sqr(
#define sqr(x) ((x)*(x))
...
int i = start 3 );
```

Our definition does not permit a macro body to contain unmatched parentheses, if these parentheses follow a macro name whose replacement is intended. The formal derivation is:

```
    f{}("int i = start 3);",)
=   "int i = " || f{start}(Rep("start")) || f{}("3);")
=   "int i = " || f{start}("sqr (") || "3);"
=   "int i = sqr(3);"
```

The result draws a syntax error from a language processor unless Sqr is a defined function.

But it is still possible to construct a macro invocation from pieces when the pieces are joined within the replacement text of a single macro:

```
#define x g(h
#define h(f) f(3))
#define g(i) i+1
h(x);
```

```
    f{}("h(x);")
=   f{h}(Rep(h,f{}("x")))                  || f{}(";")
=   f{h}(Rep(h,f{x}(Rep("x"))))            || ";"
=   f{h}(Rep(h,f{x}("g(h")))               || ";"
=   f{h}(Rep(h,"g(h")))                    || ";"
=   f{h}(f{h}("g(h(3))"))                  || ";"
=   f{h}(f{h,g}(Rep(g,f{h}("h(3)"))))      || ";"
=   f{h}(f{h,g}(Rep(g,"h(3)")))            || ";"
=   f{h}(f{h,g}("h(3)+1"))                 || ";"
=   f{h}("h(3)+1")                         || ";"
=   "h(3)+1"                               || ";"
=   "h(3)+1;"
```

Here an invocation of h was constructed from the argument x, expanded to g(h, and the matching parentheses in the body of h. The best one can say about this usage is that it is good for boundary examples in documents such as this one.

The C compilers mentioned previously generate "3(3))+1" from "h(x)", mainly because the way they avoid an endless loop is by placing an upper limit on a macro expansion buffer, rather than actually detecting the loop.

5.9 Predefined Macros

[__LINE__, __FILE__]

__LINE__ and __FILE__ are predefined parameterless macros. The first expands to a decimal digit string whose value is the line number of the source file containing the occurrence of __LINE__. The second expands to a quoted string literal, the name of the file containing the occurrence of __FILE__. These macros are most useful for debugging:

```
printf("Now at line %d in file %s\n",
        __LINE__, __FILE__);
```

5.10 Conditional Inclusion

[#if, #ifdef, #ifndef, #elif, #else, #endif; defined; constant expression; macro replacement; enabling condition; Signed-Long-Int]

Syntax

```
Control_text
-> ( '<CONTROL>'!'if'    E '<C-EOL>' If:Words
   | '<CONTROL>'!'ifdef'  '<IDENTIFIER>' '<C-EOL>' Words
   | '<CONTROL>'!'ifndef' '<IDENTIFIER>' '<C-EOL>' Words
   )
    ('<CONTROL>'!'elif'  E '<C-EOL>' Elif:Words)*
    ('<CONTROL>'!'else'    '<C-EOL>' Else:Words)?
     '<CONTROL>'!'endif'   '<C-EOL>'
;
E -> ... see the next paragraph.
```

The E nonterminal generates the same language as E2 in the C phrase-structure grammar (see Section *Expressions*), except that the only allowable occurrences of <IDENTIFIER>s in E are as follows:

(i) '<IDENTIFIER>'!'defined' Mname:'<IDENTIFIER>'
(ii) '<IDENTIFIER>'!'defined' '(' Mname:'<IDENTIFIER>' ')'
(iii) Mname:'<IDENTIFIER>'

Constraints

E must generate a constant expression (see Section *Expressions/Constant Expressions*) of an integral type with no occurrences of the **sizeof** operator, enumeration constants, or type casts.

Semantics

Macro replacement does not occur for the single word appearing after '<IDENTIFIER>'!'defined', in case (i) or (ii), or for the single word appearing after the '(' in case (ii). In addition, no replacement occurs for the <IDENTIFIER> following **#ifdef** or **#ifndef**.

Conditional inclusion. **#if-#ifdef-#ifndef**, **#elif**, and **#else** provide a conditional text inclusion facility based upon the value of an expression E or the existence of a macro definition. Only one of the sequences of Words in a **#ifxxx-#endif** command is included; all others are excluded.

The included sequence is the first sequence of Words such that the *enabling condition* of the preprocessor command containing the Words is True. The enabling condition for the **#if** and **#elif** commands is that E evaluates to a non-zero integer value. The enabling condition for the **#ifdef** command is that the <IDENTIFIER> is defined as a macro; for the **#ifndef** it is the reverse. The enabling condition for the **#else** command is True, i.e. if no other enabling condition holds, the Words associated with the **#else** command are included.

Macro replacement does not occur on excluded text.

Evaluation of E. Each conditional inclusion expression E is evaluated using the host environment's Signed-Long-Int arithmetic, as if all operands had type Signed-Long-Int rather than the type normally dictated by the Constraints of C expressions; see Section *Expressions.*

In all cases (i)-(iii) for <IDENTIFIER>s, the expression evaluates to zero or one according to whether Mname is currently **#define**-d or not. Note that in the third case of a single <IDENTIFIER>, the value one is rarely obtained, since the only way the <IDENTIFIER> could be a macro name is if the macro were recursively defined so that replacement stopped, leaving the macro name. *Examples:*

```
#define x x
#if x       /* Same as "if 1". */
   ... this text is included ...
#endif

#undef x
#if x       /* Same as "if 0". */
   ... this text is excluded ...
#endif
```

Obsolescence. **#ifdef** and **#ifndef** provide nothing that **#if-#elif-#else-#endif** do not already provide, and the former's use is discouraged in favor of the latter's. **#ifdef** and **#ifndef** can be avoided as follows:

This	*can be replaced with*
#ifdef x	**#if defined**(x)
. . .	. . .
#endif	**#endif**
#ifndef x	**#if !defined**(x)
. . .	. . .
#endif	**#endif**

Some compilers permit arbitrary text on a line following **#endif** and **#else**. This supports a common UNIX practice exemplified by "**#ifdef** X ... **#endif** X". X3J11 recently decided

to permit the text after **#endif** and **#else** to be implementation-defined.

5.11 Preprocessor Words

Syntax

```
Words -> Word*;
Word  -> Any - '<CONTROL>' - '<C-EOL>'
       # Any word produced by lexical analysis
       # other than the two explicitly excepted.
```

Semantics

Macro replacement occurs on all Words, with the exceptions as noted above: for the word following '<IDENTIFIER>'! 'defined' and the word following '<IDENTIFIER>'!'defined' '(' in the case of the conditional inclusion expression E, for excluded Words, and for Words that are a macro Body.

6

Declarations

6.1 External Declarations

[Unspecified_declaration; Function_definition; Non_function_definitions; Specified_declaration; pragmas from Ada; toggles; block; #pragma]

Syntax

```
parser C_phrase_structure:
C_phrase_structure
-> External_declaration*
;
############################################################
# Declarations.                                            #
############################################################
External_declaration
-> Unspecified_declaration:
   ( Function_definition
   | Non_function_definitions ';'
   )
-> Specified_declaration        # With specifiers.
-> Pragma_call
-> ';'                          # Syntactic oddity of KR.
;
Pragma_call
-> 'pragma' Name  ('(' (E list ',')? ')')? ';'
;
```

A C program consists of a sequence of declarations and pragmas. Some declarations are also definitions.

Constraints

There is a block E that is the text of the entire program. Any block introduced by any definition — an Unspecified_definition or a Specified_declaration — is a part of E.

Specifiers are assumed in an Unspecified_declaration according to rules set forth in the next Subsection.

Semantics None.

Discussion

Pragmas are found in no other definition of C. The syntax is taken from Ada. The intent is to allow the programmer to give directives to a C language processor. For example, all MetaWare C compilers support pragmas named Set, Reset, and Pop, which take a single identifier as the name of a *toggle stack* and either push True (set), push False (Reset), or pop the stack. The value on the top of the toggle stack affects language translation in some way, depending upon the nature of the toggle.

Beyond this we say no more about pragmas, leaving their further definition to another document more specific to MetaWare High C and High C implementations. A related notion is the newly-introduced "**#pragma**" preprocessor command found in the X3J11 document; we have not included **#pragma** as part of our definition.

Pragmas are also permitted in statements; see Section *Statements:*

6.2 Specified Declarations

[Specifiers; Function_definition; Non_function_definitions; Specified_declaration; External_declaration; automatic storage class]

Syntax

```
Specified_declaration
-> Specifiers ';'
-> Specifiers Function_definition
-> Specifiers Non_function_definitions ';'
;
```

Constraints

The Specifiers of a Specified_declaration that is syntactically immediately derived from an External_declaration may not denote the storage class automatic; see 6.3 below for how Specifiers denote a storage class.

Semantics None.

Discussion

Declarations are broken up into (1) the definition of functions (Function_definition) and (2) declarations that are not function definitions (Non_function_definitions), but that may be function declarations or definitions of names of non-function types.

Only one function can be declared in a Function_definition, but many non-functions may be declared in Non_function_definitions.

The alternative "Specifiers ';'" is permitted to allow the declaration of a type T without also declaring a name for type T. For example, "**struct s {int x;}; typedef int t;**" declares a tag **s** referring to the Struct{...} type and a **typedef** name t standing for the Signed-Int type.

6.3 Types and Specifiers

[Storage_classes: auto, extern, register, typedef, static; adjectives: short, unsigned, long, signed; char, int, float, double, void; <TYPEDEF_NAME>; Specifiers; Type_specifiers; Compound_statement; Non_function_definition; Function_definition; Unspecified_declaration; Specified_declaration; static-import, static-export, static-private; automatic, typedef; type of char; arithmetic types]

Syntax

```
Specifiers
-> Type_or_storage_classes
;
Type_or_storage_classes
-> Storage_class Type_or_storage_classes?
-> Typedef_reference:'<TYPEDEF_NAME>' Storage_class?
-> Type ASCs
-> Adjective ASCs (Type ASCs)?
;
ASCs -> (Adjective | Storage_class)*
;
Storage_class
-> 'auto'|'extern'|'register'|'typedef'|'static'
;
```

```
Adjective
-> 'short'|'unsigned'|'long'|'signed'
;
Type_specifiers
-> Typedef_reference: '<TYPEDEF_NAME>'
-> Adjective* Type Adjective*
-> Adjective+
;
Type
-> <'char'|'int'|'float'|'double'|'void'>
-> Tagged_type;
```

Specifiers are complicated by the syntactic rules for the interpretation of <IDENTIFIER>s as <TYPEDEF_NAME>s. The complexity bears some detailed explanation here.

Specifiers are essentially a sequence of Storage_classes, Types, Adjectives, and <TYPEDEF_NAME>s, where

(a) there may be at most one Storage_class, a constraint that would be clumsily imposed by the grammar;

(b) there may be at most one Type or <TYPEDEF_NAME>: imposed by the grammar, and necessary to properly interpret <IDENTIFIER>s as <TYPEDEF_NAME>s where appropriate — this is the reason for the grammar's complexity;

(c) a <TYPEDEF_NAME> may not be combined with an Adjective;

(d) only certain combinations of Adjectives are permitted, a constraint that would be clumsily imposed grammatically.

A <TYPEDEF_NAME> is an <IDENTIFIER> that is declared of mode **typedef** in the ordinary name space. Effectively, any time such an <IDENTIFIER> word appears, it becomes a <TYPEDEF_NAME> word. The grammar permits at most one occurrence of a <TYPEDEF_NAME> in Specifiers or Type_specifiers, and does not allow it to be combined with a Type. Essentially, the <TYPEDEF_NAME> specifies the type (see *Constraints* below), and therefore the combination with another Type, <TYPEDEF_NAME>, or Adjective (type modifier) is meaningless. The ice is *not* thin here.

Examples:

```
typedef int T;
void F() {
   T x;          /* x is of type T = Signed-Int.          */
   short T x;    /* Illegal: adjective not allowed.       */
   T T x;        /* Illegal: two <TYPEDEF_NAME>s.         */
   register T;   /* Illegal: declared name missing.       */
   int T;        /* A valid re-declaration of T.          */
   }
```

The nonterminal Type_specifiers is the subset of Specifiers that forbids Storage_classes, and is used elsewhere in the grammar.

Constraints

Specifiers and Type_specifiers denote a type and a storage class that become associated with the name declared with the Specifiers. How to determine that type and storage class is discussed next.

Specification of the storage class. In Specifiers there may be at most one Storage_class stated. If the Storage_class is omitted, a storage class is implied based upon where the specified declaration appears, as follows:

(a) In an Unspecified_declaration, a Function_definition has storage class static-export; a Non_function_definition of mode fcn has storage class static-import; any other Non_function_definition has storage class static-export.

(b) In a Specified_declaration that is a part of an External_declaration, the rules in (a) above apply.

(c) In a Specified_declaration that is a part of a Compound_statement, a Function_definition has storage class automatic; a Non_function_definition of mode fcn has storage class static-import; any other Non_function_definition has storage class automatic.

Examples:

```
   /* Illustrations of case (a):                                    */
x;              /* Unspecified_declaration:Non_function_definition
                   => static-export storage class.                  */
y();            /* Unspecified_declaration:Non_function_definition
                   => static-import (due to mode fcn).              */
f(){            /* Unspecified_declaration:Function_definition
                   => static-export storage class.                  */

   /* Illustrations of case (c):                                    */
   void g(){...} /* Specified_declaration that is part of a
                    Compound_statement => automatic.  But this
                    particular case is ruled out elsewhere --
                    nested functions are illegal.                   */
   int h();     /* Specified_declaration that is part of a
                   Compound_statement => static-import.             */
   int i;       /* Specified_declaration that is part of a
                   Compound_statement => automatic.                 */
   }

   /* Illustration of case (b):                                     */
int z;          /* Specified_declaration that is a part of an
                   External_declaration => static-export.           */
```

End of Examples.

If the Storage_class is stated:

Storage_class	*denotes storage class*
extern	static-import
static	static-private
auto or **register**	automatic
typedef	typedef

Note. The static-export storage class *cannot* be explicitly written, but is the default when none is specified.

The Storage_class **auto** is allowed only in a declaration within a function; **register** is allowed in the same circumstances and additionally as the storage class of function Parameters.

Specification of the type. In Specifiers and Type_specifiers, the grammar permits only zero or one Types or <TYPE-

DEF_NAME>s. Any number of Adjectives is permitted by the grammar, but only certain combinations are legal. In any particular combination the Adjective(s) and/or Type may be presented in any textual order, but no Adjective may be repeated. No Adjective may appear with a <TYPEDEF_NAME>.

If a <TYPEDEF_NAME> is used, it it guaranteed to be declared of mode **typedef** in the ordinary name space of a type T. The type denoted by the Typedef_reference is T and is the specified type.

If a <TYPEDEF_NAME> is not used. The next table indicates the only allowable combinations of Adjectives with **int** and **char**, and the correspondingly denoted type. The first three rows may be optionally combined with the Type **int**.

Table Adjectives Combined with **int** and **char**.

	signed or nothing	**unsigned**	
short	Signed-Short-Int	Unsigned-Short-Int	with
long	Signed-Long-Int	Unsigned-Long-Int	or without
(nothing)	Signed-Int	Unsigned-Int	the Type **int**.
char	Signed-Char†	Unsigned-Char	
	denoted type		

†*Exception:* whether **char** standing alone denotes Unsigned- or Signed-Char is implementation-defined.

Therefore, if Specifiers contains neither a Type nor an Adjective, the "(nothing)-nothing" intersection in the table above applies and the denoted type is Signed-Int.

The Types **float**, **double**, and **void** standing alone denote the types Float, Double, and Void. **double** may be combined with **long** to denote the type Long-Double. No other Adjective may be combined with **double**, and **float** and **void** may not be combined with Adjectives at all.

For example, "**unsigned int short**" and "**int short unsigned**" are both allowed and denote the type Unsigned-Short-Int, but "**int short unsigned short**" is not allowed,

since **short** occurs twice.

Tagged_types are covered in the next Subsection.

```
   Examples:                /* Storage class: Type:                    */
x;                          /* Static-export  Signed-Int               */
f();                        /* Static-import  (?) -> Signed-Int        */
char y, z;                  /* Static-export  Signed- or Unsigned-Char */
static q, r;                /* Static-private Signed-Int               */
main() {                    /* Static-export  () -> Signed-Int         */
   unsigned x, y;           /* Automatic      Unsigned-Int             */
   auto short z, w;         /* Automatic      Signed-Short-Int         */
   double extern long l;
                            /* Static-import  Long-double              */
   }
```

Semantics

The meaning of the type specifications and storage classes is discussed in the Section *Concepts.*

Discussion

- The reason that the type T denoted by **char** standing alone is implementation-defined as either Unsigned-Char or Signed-Char is due to potential inefficiences in a computer architecture. For example, on the IBM 370 loading an Unsigned-Char is much more efficient than loading a Signed-Char, so T = Unsigned-Char is preferable. On an 8088, widening an Unsigned-Char to a Signed-Int costs a two-byte instruction, whereas widening a Signed-Char to a Signed-Int costs only a single byte.
- Although a Function_definition of storage class automatic is not ruled out by the constraints in this subsection, they are in Subsection 6.5.
- Formally, the Storage_classes **auto** and **register** have the same semantics: specification of the automatic storage class. Generally the Storage_class **register** is understood by a language processor to place the declared object's value in a register; however *no semantic change to the program* must occur through this placement.

Example: Some implementations, e.g. Microsoft C 3.00 on the 8086 under MS-DOS, do not truncate unsigned character values when the character is held in a register, so that in "**register char** c = 255; **if** (++c == 256) f();" f may well be called on an 8-bit-byte machine. The value of c after the increment should instead be zero.

- The only combination of the Types and Adjectives allowed here and in KR are "**short int**", "**long int**", and "**unsigned int**". It is not clear from KR whether the order matters. KR also permits "**long float**" to stand for "**double**"; High C, like X3J11, does not. High C, like X3J11, uses "**long double**" to denote type Long-Double, a type not in KR.

- 4.2BSD permits Adjectives to appear with <TYPEDEF_NAME>s. The result is somewhat as if <TYPEDEF_NAME>s were macros instead of names associated with a distinct type. *Example:* "**typedef int** T; **unsigned** T x; T y;" is permitted.

  This has the perhaps unusual effect that x and y are *not* of the same type; x is of type Unsigned-Int and y of type Signed-Int. The Adjective overrides the type associated with T, changing the type from Signed-Int to Unsigned-Int.

  Note the distinction: "**int**" standing alone denotes the signed integer type, not an "incomplete" integer type; specific syntactic combinations to denote various integer types are listed above. Both KR and X3J11 concur with this manual in insisting that a **typedef** name denote a *single* unmodifiable type.

- Many C implementations are two-faced about the use of the Storage_class **extern**. For a function declaration that is not a definition, i.e. the body is not being supplied, **extern** always means the function is declared elsewhere, but **extern** on a function definition denotes storage class static-export.

High C reserves extern for static-import only, and formally forbids it on definitions. In a concession to existing (poor) practice, MetaWare High C compilers warn when it appears on a function definition. Its presence is unnecessary since the absence of any Storage_class for a function definition implies storage class static-export.

6.4 Tagged Types

[struct, union, enum; incomplete structure or union type; tag name space; structure or union members; bit field; enumeration literals and type; member alignment]

Syntax

```
Tagged_type
-> Complete_definition: 'struct' Tag? '{' Member_list '}'
-> Complete_definition: 'union'  Tag? '{' Member_list '}'
-> Use_or_incomplete_definition: ('struct'|'union') Tag
-> Complete_definition: 'enum'  Tag? '{' Literal_list '}'
-> Reference:  'enum' Tag
;
Literal_list
-> (Name ('=' Constant:E)?) list ',' ',?
;
Tag
-> Tag:Name
;
Member_list
-> '{' Also_is_a_list:Members list ';' ';'? '}'
;
Members
-> Type_specifiers (Structure_member list ',')?
;
Structure_member
-> Declarator
-> Field_member:Declarator? ':' Bits:Constant:E
;
```

Constraints

Structures and Unions. If the Tag (Name) is given in a Complete_struct_definition:

(a) if the Tag is declared of mode **struct-tag** and incomplete type Struct{?} in the tag name space, and the origin of the latter declaration is the same as that of the Complete_struct_definition, then the Member_list specification completes the original definition of the incomplete **struct**. From this point through the rest of the origin of the Tag's declaration, the incomplete **struct** has type Struct{M}, where M is the Member_list. This does *not* create a new type but merely completes an old one. The type denoted by the Complete_struct_definition is this completed type.

Note that the requirements of "same origin" prohibit the example cited in Section *Introduction*, a particular sore spot for most compilers and C language definitions.

(b) Otherwise, the occurrence of the Tag is its defining-point within the tag name space of mode **struct-tag**; it has new type T = Struct{?} from its defining-point to the closing } of the definition, and then has type Struct{M} where M is the Member_list; the type denoted by the Complete_struct_definition is this new type.

Furthermore, T may not be completed within the definition. This unusual two-stage type association prevents illegal declarations such as "**struct** S{**struct** S x;}", but permits such declarations as "**struct** S{**struct** S *x;}": pointers to incomplete types are permitted.

If no Tag is given in a Complete_struct_definition, the definition denotes a new type Struct{M}.

Similar constraints apply for Complete_union_definition: replace **struct** by **union** and Complete_struct_definition by Complete_union_definition in the preceding three paragraphs.

Example:

```
struct s;     /* A: Declaration of an incomplete type.  */
main() {
   struct s *y;          /* Reference to declaration A.  */
   struct s {int z;};    /* B: Duplicate declar'n of s.  */
   }
struct s {int z;};       /* C: completes declaration A.  */
```

Declaration B is not a completer of A since B and A have different origins. Therefore the occurrence of s in B is a defining-point, so that the scopes of A and B overlap, producing an illegal duplicate declaration. On the other hand, the origin of A and C is the same, so that C completes A.

No member in a Member_list may be of a functionality type or of an incomplete type (this prohibits the declaration of x in "**struct** S{S x;}".)

In a Use_or_incomplete_definition, if the Tag following **struct** is declared of mode **struct-tag** and incomplete type Struct{?} or the Tag following **union** is declared of mode **union-tag** and type Union{?}, the type denoted by the Incomplete_definition_or_use is this type. Otherwise, the occurrence of the Tag is its defining-point of mode **struct-tag** and new type Struct {?} (or of mode **union-tag** and new type Union {?}).

The type denoted by the Use_or_incomplete_definition is this type. (This means that in "**struct** S {**int** X;}; **void** F() {**union** S; ...}", the second occurrence of S is an incomplete definition, not a reference to the first S.)

The declaration of an object of type Struct{?} or Union{?} whose storage class is not static-import is illegal; the Struct or Union type must first be completed.

The E appearing in a Field_member must be a constant integral expression. If the value of E is zero, the Declarator may not be present. The Declarator must be of type Unsigned-Int or Signed-Int.

Enumerations. Each of the Constants in a an Enum_definition's Literal_list must be a constant expression of an integral type.

The type T denoted by an Enum_definition is one of Signed-Char, Unsigned-Char, Signed-Short-Int, Unsigned-Short-Int, Signed-Int, or Signed-Long-Int.

An implementation may choose any of these types for T provided that: (a) the unsigned types can be used only if their size is less than that of Signed-Int, so T widens to Signed-Int in an expression; (b) T contains the set of values specified in the Literal_list, except that a value not contained in Signed-Long-Int is converted as if by type casting to Signed-long-Int. See *Semantics* below for the specification of the values.

Generally an implementation chooses the type T having the smallest size satisfying the constraints, and chooses the signedness depending upon the efficiency of the architecture; see the *Discussion* in Subsection *Types and Specifiers* above for architectural examples.

If the Tag is provided, its occurrence is its defining-point within the Tag name space of mode **enum-tag** and the type T as specified in the previous paragraph. The occurrence of the Names in the Literal_list are their defining-points within the ordinary name space with mode **value** and type T.

In an Enum_reference, the Tag must be declared in the tag name space of mode **enum-tag** and of some type T. The type denoted by the Enum_reference is T.

Semantics

Structures and Unions. A value of type Struct{M} or Union{M} where M is a Member_list consists of a sequence of optionally named objects called members. The members and any names are defined by the Structure_member syntactic category.

The members may be either Declarators or *fields*, which are sequences of bits. The number of bits in a field is specified

by the constant expression E following the ':'. A field F that immediately follows another field F' may be placed adjacent to F' in the same storage unit, if possible.

We deliberately leave unspecified whether adjacent means that the least-significant bit of F is adjacent to the most-significant bit of F', or the least-significant bit of F' is adjacent to the most-significant bit of F, but we require that adjacent always mean the same thing in a particular implementation.

Adjacency depends upon the the order of allocation of fields within a storage unit (left-to-right or right-to-left), which is implementation-defined. An implementation may refuse to allow a field to straddle an implementation-defined storage unit boundary B.

The width of a field may not exceed the size (in bits) of an object of type Unsigned-Int.

A field of zero bits prevents any further fields from being packed into the unit of storage in which the previous field was placed, and may additionally cause the next field to be allocated on the storage unit boundary B mentioned above.

An unnamed `Field_member` (the `Declarator` is omitted) is used to conform to layouts imposed externally.

Within an object of a Struct type, the non-field members and the storage in which fields reside have addresses that increase as their declarations are read from left to right. Each non-field member of a structure may be aligned as appropriate to its type; there may therefore be unnamed gaps within a structure.

However, the address A of a structure must be the same as the address of its first member M, if M is not a field; otherwise A must be the address of the storage unit in which M is stored. Thus, there is no "gap" at the beginning of a structure.

Within an object O of a union type, the non-field members and the words in which fields reside have the same address,

which is the same as the address of 0. Thus, there is no "gap" at the beginning of a Union.

Enumerations. When used in an expression, the successive Names in a Literal_list evaluate to 0, 1, 2, ..., etc., except that a Name with an associated Constant expression E "resets" this sequence to the value of E. The values of enumeration literals need not be unique.

For example, "`enum {red, orange, blue = 0, green};`" is permitted: `red` and `blue` evaluate to zero and `orange` and `green` to one. In this example the type denoted is Signed-Char.

Discussion

4.2BSD and KR prohibit structures and unions from having members of a functionality type. X3J11 is silent on the subject.

4.2BSD permits a structure or union type T in a nested block to complete a corresponding structure or union type T' in an outer block, with confusing results. In the nested block T may be used appropriately; in the outer block, T' is incomplete and may not be completed. The compiler gives confusing messages when the use or completion of T' is attempted in the outer block.

4.2BSD warns when enumeration values and values of other types are mixed in expressions. The proposed standard and this document say that enumeration values are not of a separate type, but are of one of the basic integral types.

4.2BSD does *not* permit redefinition of **typedef** names, such as "`int x;`", where x is a **typedef** name. This appears to be a bug, since an example specifically documented in the KR book as correct is disallowed by 4.2BSD.

The address operator & may not be applied to a field member, because most computer architectures do not support addressing at the bit level. This restriction is imposed through field having the mode **field**; see the discussion of & in Section *Expressions/Pointer Reference.*

6.5 Declarators

[Parameters; parameter names and types; Abstract_parameters; Abstract_declarator; Specifiers; the type of a declarator or abstract declarator; functionality types and prototype functionalities; incomplete types; register parameters; pointer and array types; type names as parameters]

Syntax

```
Declarator
-> '*' Declarator
-> Declarator'
;
Declarator'
-> Declarator' '[' Array_specification ']'
-> '(' Declarator ')'
-> Function_specification:
   Declarator' '(' Parameters ')'
-> Declared:Name
;
Array_specification
-> Constant:E?
;
Parameters
-> Parameter_names_only:
   Parameter_name list ',' More_parms?
-> Abstract_parameters
;
Abstract_parameters
-> (SD list ',' More_parms?)?
;
SD  -> Specifiers (Abstract_declarator | Declarator2)?
;
More_parms  -> ',' '...'?;
;
Parameter_name -> '<IDENTIFIER>'
;
Abstract_declarator
-> '*' Abstract_declarator?
->     Abstract_declarator'
;
```

```
Abstract_declarator'
-> Abstract_declarator'? '[' Array_specification ']'
-> Abstract_declarator'? '(' Abstract_parameters ')'
-> '(' Abstract_declarator ')'
;
# Declarator2(') is needed to avoid an ambiguity.
Declarator2
-> '*' Declarator2
-> Declarator2'
;
Declarator2'
-> Declarator2' '[' Array_specification ']'
-> '(' Declarator2 ')'
-> Function_specification:
   Declarator2' '(' Parameters ')'
-> Declared_name: '<IDENTIFIER>'
;
```

The nonterminals Declarator2 and Declarator2' are needed to avoid a syntactic ambiguity by forbidding the name of a function's parameter to be a name previously declared of mode **typedef**; see **Type Names as Parameters** in *Discussion* below.

Constraints

Every Declarator declares a single Declared_name N. The occurrence of N in the Declarator is its defining point of a particular mode M and of a particular type T in one of two name spaces. If the Declarator is a Structure_member, the name space is that for the structure or union type having that Structure_member. Otherwise, it is the ordinary name space.

The occurrence of any Declarator is associated with a storage class S that is specified in the Type_specifiers or Specifiers preceding the Declarator, or is implied when there are no Type_specifiers or Specifiers; see Subsection *Types and Specifiers* above.

Likewise, the occurrence of any Declarator is associated with a type T' that is specified in the aforementioned Type_specifiers or Specifiers, or is implied when they are lacking. From this type T' and the structure of the Declarator, the

final type T of the declared name can be determined. We also say that the Declarator is of that type. T is *Type*(T',D), where *Type* is defined recursively as follows:

Table Declarator Type Determination.

Type(T,D) = *if D is of the form*	*then*
*D'	*Type*(*T, D')
D' [ ]	*Type*([?]:T, D')
D' [E]	*Type*([V]:T, D') where E must be a Constant expression of an integral type whose value is V
(D')	*Type*(T, D')
D' Parameters	*Type*(F, D') where the determination of the functionality type F is deferred to later discussion under the heading **Function Parameters**
Name	T

Examples:

```
int *(*x)[3], (*f(int))[];
```

x's type is determined as follows:

Type(Signed-Int, *(*x)[3])
= *Type*(*Signed-Int, (*x)[3])
= *Type*([3]:*Signed-Int, (*x))
= *Type*([3]:*Signed-Int, *x)
= *Type*(*[3]:*Signed-Int, x)
= *[3]:*Signed-Int

i.e. x is a pointer to an array of pointers to Signed-Ints. f's type is determined as follows:

Type(Signed-Int, (*f(int))[])
= *Type*([]:Signed-Int, (*f(int)))
= *Type*([]:Signed-Int, *f(int))
= *Type*(*[]:Signed-Int, f(int))
= *Type*((Signed-Int)$_p$->*[]:Signed-Int, f)
= (Signed-Int)$_p$->*[]:Signed-Int

i.e. f is a function (with prototype functionality) taking a Signed-Int; it returns a pointer to an array of Signed-Ints.

Abstract_declarator provides a way of specifying a type without declaring a name of that type. Like Declarator, Abstract_declarator appears in a context that associates a type T' with it; in the rule for Abstract_parameters, the type comes from the preceding Specifiers, and in the rule for Cast_type (see Section *Expressions/Cast Types and Abstract Declarators*), from a preceding Type_specifiers.

The type T denoted by an Abstract_declarator A is *Type*(T',A), defined recursively as follows:

Table Abstract_declarator Type Determination.

Type(T,A) = *if A is of the form*	*then*
*	*T
*A'	*Type*(*T, A')
[]	[]:T
A' []	*Type*([]:T, A')
[E]	[V]:T
A' [E]	*Type*([V]:T, A')
where E must be a Constant expression of an integral type whose value is V	
Abstract_parameters	F->T
A' Abstract_parameters	*Type*(F->T, A')
where the determination of the functionality type F is deferred to later discussion under the heading Function Parameters	
(A')	*Type*(T, A')

Mode of declared names. Each declared name has a mode, determined as follows.

If the Declarator is a Structure_member:

The mode of the name is	*if*
field	the name is a (bit) field of the structure;
member	otherwise.

If the Declarator is not *a Structure_member:*

The mode of the name is	*if*	
typedef	the storage class is	typedef;
fcn	the storage class is not typedef and T is a functionality	type;
var	otherwise.	

Examples:

```
static int e;          /* Mode var.      */
extern int f();        /* Mode fcn.      */
typedef int (*g)();    /* Mode typedef.  */
struct {
   int h;              /* Mode member.   */
   int i:5;            /* Mode field.    */
   } j;                /* Mode var.      */
static int k() {}      /* Mode fcn.      */
static int (*l)();     /* Mode var.      */
```

Incomplete types. A Declarator of an incomplete type must have storage class static-import and mode **var**, unless it is accompanied by an Initializer that completes the type; see Subsection 6.7 below. For example, "**extern struct** s x; **extern int** a[];" (where s is undeclared in the tag name space) is permitted, but "**struct** s x; **int** a[];" is not. In the latter case the sizes of x and a are necessary for storage allocation but cannot be determined.

Note. Since each structure member is of mode **member** or **field**, this rule effectively prohibits members having incomplete types. Since parameters are of automatic storage class, they too may not be of an incomplete type (but see below, where the apparent incomplete type [?]:T for a parameter is actually *T, so that the parameter declaration "**char** c[];" is legal). But pointers to incomplete types are always permitted. The essential issue is that the size of an object of an incomplete type is unknown.

Array types. The types [?]:T and [V]:T where T is an incomplete type are not permitted.

Void types. No Declarator may have type Void. However, an Abstract_declarator may have type Void.

Functions. A Declarator of mode **fcn** that is not part of a Function_definition may *not* have storage class static-export or static-private. For example, the declaration "**static g();**" is not permitted.

A Declarator of mode **fcn** may not have its type determined by a **typedef** of a functionality type. Although there is no implementation difficulty in this case, the restriction helps prevent confusing programs.

Examples:

```
typedef f(int x);
extern f g;          /* Illegal.                                  */
extern f h {         /* Illegal.                                  */
   x = 1;            /* Were h legal, this would be a confus-     */
   }                 /* ing statement; where does x come from?    */
f *pf;               /* But this is legal.                        */
main() {
   (*pf)(3);
   }
```

Function parameters. Parameters is syntactically of two different forms: Parameter_names_only and Abstract_parameters. In the latter case the types of the parameters can be given through Specifiers, and some or all of the parameter names may be given if the Declarator2 alternative for each parameter is used; see the rule for Abstract_parameters. Additionally the "()" case of parameter specification is permitted through Abstract_parameters.

In the Parameter_names_only case, only the parameter names are given, and the types of the parameters are left unspecified until the completion of the Function_definition containing the Parameters; see *Function Definitions* below.

Therefore there are three possibilities for a parameter: the parameter name only; the parameter type only; and the parameter name and type. No matter how the parameter names are provided, they must be distinct from each other.

Now to the deferred discussion of how to determine F in the Type Determination tables above. F is one of the four functionality type classes discussed in Section *Concepts*.

If Parameters is simply "()" (via Abstract_parameters), F is (?)->T if D *is not* part of a function definition, and ()->T if D *is* part of a function definition.

If Parameters is a parenthesized list of parameter names (via Parameter_names_only), F is $(T_1,...T_n) \rightarrow T$, where the T_i are the types of the parameters as declared in subsequent Parameter_types in the completion of the Function_definition.

If Parameters is a parenthesized list of one or more parameter types and optionally names (via Abstract_parameters), F is $(T_1,...T_n)_p \rightarrow T$, where the T_i are the parameter types. If the list includes "..." at its tail, F is instead $(T_1,...T_n,...)_p \rightarrow T$.

Finally, if Parameters is "(void)", F is $()_p \rightarrow T$.

The types $(T_1,...T_n)_p \rightarrow T$ and $(T_1,...T_n,...)_p \rightarrow T$ are known as *prototype functionalities*. The semantics of function call for functions of prototype functionality is quite different from those without such; see Section *Expressions/Function Call* where the difference surfaces. Prototypes fill a badly needed void in "old C": a way to provide checkable type information for imported functions.

The type P -> T where T is an array, functionality, or incomplete type is not permitted.

If the Declarator is part of a Function_definition, Parameters and the Parameter_types and the Compound_statement of the Function_definition are together a block. If the Declarator is not part of a Function_definition, Parameters is a block. See *Function Definitions* below.

The only storage class permitted in the Specifiers of Abstract_parameters is **register**.

In Abstract_parameters, the types of the parameters are always given by using Specifiers in conjunction with Abstract_declarator or Declarator2. When Abstract_declarator is used, only the parameter type is specified; when Declarator2 is used, the Declared_name of the declarator is the parameter name.

> *Example:* "**int h(float, int x);**" gives a prototype functionality for function f where only the second parameter is named. Abstract_declarator has been used for "**float**", and Declarator2 for "**int x**". In this example, naming only one parameter does not make sense, though it is allowed. When both parameters are named, an extension known as *named parameter association* (from Ada) is possible; see Appendix *Extensions*.

A parameter declared of type [?]:T or [V]:T is not of those types, but is of type *T. A parameter of a functionality type is not permitted.

(The Constraints and Semantics for function calls — in particular, the type matching of actual to formal parameters — are presented in Section *Expressions/Function Call*.)

Semantics

Pointer types. An object of type *T is capable of holding the address of any object of type T. All pointer types occupy the same amount of storage.

Array types. An object O of type [V]:T is a sequence of V objects of type T indexed from 0..V-1. The size of O is V*(**sizeof** T). The address of O is the same as the address of O[0]; in general, &O[I+1] = &O[I] + 1 = the address of O[I] plus **sizeof**(T) storage units, i.e. array elements are contiguous.

An object of type [?]:T is a sequence of unknown length of objects of type T. Its size is unknown, but the relationship between its address and that of its elements is the same as for [V]:T.

Functionality types. A name of type P -> T refers to a function returning type T. See *Function Definitions* below for more information.

Discussion

- An object of mode **var** can be assigned values. An object of mode **fcn** is not a variable, but labels function code and thus cannot be altered. See Section *Expressions/Assignments*, where only objects of mode **var**, **field**, or **member** can be assigned.
- 4.2BSD is confused about "**static int** F1();". Calls to F1 become calls to an external function; it is as if the declaration read "**extern int** F1();". But the function may not be defined in the source text containing the declaration; hence it is as if "**extern**" were *not* written. By contrast, "**extern int** F2();" declares an external function that can be later defined; this is proper. As a declaration local to a function, 4.2BSD allows "**static int** f();" to mean exactly "**extern int** f();".

  4.2BSD also allows the declaration "**extern int** f() {}" which essentially says that f is declared elsewhere ("**extern**") but is actually declared right here ("{}")! X3J11 requires another declaration of f() elsewhere without "**extern**".

  Basically, C is somewhat confused with respect to importation and exportation and the use of **extern**. High C interprets **extern** strictly as "defined elsewhere".
- KR prohibits the following types: P->T, where T is a functionality, array, structure, or union type; [?]:T' or [V]:T', where T' is a functionality type; or structures or unions containing functions. Curiously, X3J11 does not impose any of these restrictions, but does not ascribe semantics to the cases where T is a functionality or array type, or where T' is a functionality type.

Parameter scoping. KR does not address the issue of the scope of parameters. X3J11 and this document agree in spe-

cifying that the parameters are contained in the same block introduced by the Compound_statement of a Function_definition. By contrast, 4.2BSD allows re-declaration of a parameter in the Compound_statement, a "feature" that can lead to agonizing bugs, as in

```
int f(a,b)
   int a; char *b;
   {
   int x; char *b;
   x = a+b;     /* Was the parameter b meant here? */
   return x;
   }
```

Type names as parameters. Note that Abstract_parameters uses the nonterminal Declarator2, which replicates the Declarator syntax except that the declared name can only be an <IDENTIFIER>, rather than a Name, which can be either an <IDENTIFIER> or a <TYPEDEF_NAME>. Likewise, Parameter_name can produce just <IDENTIFIER>. This means that a function parameter cannot be an identifier previously declared of mode typedef.

Examples:

```
typedef enum{False,True} Boolean;
void f1() {
   char Boolean;          /* is   legal.          */
   }
void f2(Boolean);         /* is   legal.          */
void f3(Bulyean);         /* is   legal.          */
void f4(int Boolean)      /* is illegal.          */
   { }
void f5(Boolean)
   int  Boolean;          /* is illegal.          */
   { }
void f6(Boolean i);       /* is   legal.          */
```

The main reason for the restriction is exemplified by the declarations f2 and f3. f3 declares an external function f3 whose parameter is named Bulyean and is of type Signed-Int. Without the restriction, f2 could be interpreted as either a

function taking a parameter named Boolean of type Signed-Int, or a function taking an unnamed parameter of type Boolean. The restriction resolves the ambiguity.

In declaration f5, even though it is clear that the parameter's name is Boolean, the information is available too late (at the second occurrence of Boolean) for proper parsing. So, even though this case is not ambiguous, the restriction eases the translator's work.

Division of labor between Declarators and Specifiers. Declarators are a way to declare a name, and optionally participate in the construction of the type of the name. Here the C language is confusing: the task of specifying the type of a declared name is divided between Declarators and Specifiers.

Specifiers permit the reference to and the construction of a structure, union, and enumeration type, or a reference to any of the basic types or a previously defined **typedef** type. Declarators permit the specification of array, function, and pointer types.

The stated reason for placing some of the type specification in Declarators is that it is possible thereby to make the declaration of a type appear similar to the use in expressions of objects of the type. Any type whose use and definition could be made similar was placed in Declarators; all other types were placed in Specifiers.

This in fact can make C declarations difficult to read, and the type of an object difficult to discern. It can also cause problems for novices: Many times this author has construed "**char*** x, y;" as the declaration of two variables of type *Signed-Char. But since the * is part of the Declarator, not the Specifier, y is of type Signed-Char. By contrast, in "**struct** {...} x, y;", both x and y are of the same aggregate type.

6.6 Function Definitions

[parameter names and types; Abstract_parameters; Specifiers; functionality types and prototype functionalities; register parameters; Function_definition; argument conversion at function entry]

Syntax

```
Function_definition
-> Function:Declarator Parameter_types Compound_statement
;
Parameter_types
-> ( Specifiers (Parameter:Declarator list ',')? ';' )*
;
```

Constraints

The Declarator D in a Function_definition must be of a functionality type P->T. The Parameters in D, the Parameter_types, and the Compound_statement, all constitute a block.

The only Storage_class allowed in the Specifiers preceding a Parameter:Declarator is **register**.

Parameters is syntactically of two different forms: Parameter_names_only, and Abstract_parameters.

Using Abstract_parameters. The Declarator2 form (see the rule for Abstract_parameters) must be used for each parameter so that the parameter name is supplied. Furthermore, Parameter_types must be empty.

Using Parameter_names_only. If there are one or more Parameter:Declarators in Parameter_types, the Declared_name N of each such Declarator must not be the same as that of any other such Declarator, and each Declared_name N must be one of the parameter names. In this case parameter N has the type denoted by the Specifiers and Parameter:Declarator. If no Parameter_type declaration is given for a parameter name p, the Parameter_type declaration "**int** p;" is implied, i.e. the parameter has type Signed-Int and storage class automatic.

Note. These rules permit other names not in the ordinary name space to be declared in the Parameter:Declarators: structure tags and member names.

Example:

```
int f(a,b)
   struct s {int z;} a;
   {...
```

Here s is declared as a tag and z a structure member, in addition to the declaration of the parameter a. Since there is no Parameter_type for b, "**int** b;" is implied. It is also possible to give just Specifiers alone with no Parameter: Declarator; e.g. "**struct** s2 {**int** w;};" is legal before the opening { of the function.

The Storage_class **extern** may not be used with a Function_definition, since it is inconsistent: **extern** implies the declaration is elsewhere, yet the function body is supplied as the Compound_statement.

Semantics

A Function_definition provides the code body for an object of a functionality type. If the storage class of the declared name N is static-export, any other declaration of a name N with mode **fcn** and storage class static-import in any other source text refers to this code body; see Section *Concepts/ Independent Translation.*

If a function F does not have prototype functionality, actual arguments corresponding to parameters of type Signed-Char, Unsigned-Char, Signed-Short-Int, Unsigned-Short-Int, or Float are assumed to have been passed as their converted counterparts according to the rules of a function call (see Section *Expressions/Function Call*), and are converted back to the declared formal parameter type at the outset of the execution of the function body.

More precisely, for a parameter of type Signed-Char or Signed-Short-Int, it is assumed that the argument is passed as

Signed-Int, and so conversion from Signed-Int to the parameter type occurs. For a parameter of type Unsigned-Char or Unsigned-Short-Int, it is assumed that the argument is passed as Unsigned-Int, and so conversion from Unsigned-Int to the parameter type occurs. For a parameter of type Float, it is assumed that the argument is passed as Double, and so conversion from Double to Float occurs.

The Storage_class **register** supplied for a function parameter is a request that the parameter be held in a register during the execution of the function; it does *not* affect the semantics of the program. A program with **register** declarations must have the same meaning as when the Storage_class **register** is arbitrarily omitted.

Discussion

Prototype functionality is new, designed to allow some type-checking across separate compilation units. If a non-defining declaration for function F precedes a function definition for F, our rules for duplicate declarations (see Section *Concepts*) require that both have identical prototype functionality if either have a prototype functionality.

X3J11 instead adopts the notion that a prototype functionality for the first declaration and a non-prototype functionality for the second are permissible, where the semantics of the second are as if it was declared with a prototype functionality:

```
int f(float x);
int f(x) float x; {
   ...
   }
void Use_f() {
   f(3.0);          /* Pass in Float, not Double. */
   }
```

The X3J11 approach ruins the readability of the function definition of f: now its semantics are different depending upon whether a prototype definition for f precedes it. High C rules require that the definition for f also use prototype function-

ality, retaining the property of "old C" that function definitions are understandable out-of-context.

6.7 Non-Function Definitions

[Non_function_definitions; Initializers and initialization]

Syntax

```
Non_function_definitions
-> (Declarator ('=' Initializer)?) list ','
;
Initializer
-> E
-> '{' Initializer list ',' ','? '}'
;
```

Constraints and Semantics

Due to the complexity of describing the initialization of objects, Constraints and Semantics are combined.

Each Declarator in Non_function_definitions may optionally be initialized. The mode of the Declarator must be **var**; thus "**extern** F() = {...}" is not permitted, because F is of mode **fcn**; nor is "**typedef** T[3] = {1,2,3};", since T is of mode **typedef**. Only objects of storage class static-export, static-private, or automatic may be initialized. If an object of storage class static is initialized, the Initializers must be constant expressions.

Assume the object O being initialized is of type T. When T is a scalar type, the Initializer consists of a single expression E, optionally enclosed in braces. When T is a Struct{...} or Union{...} type, the Initializer may be a single expression E, *not* enclosed within braces. The type of E must be assignment-compatible with T. *Semantics:* O is initialized with the value of E converted to type T.

The remaining discussion applies in all cases excluding those just detailed.

When O is an aggregate, the Initializer is a brace-enclosed list of Initializers for the members of O, written in increasing subscript or member order; there may be no more Initializers than aggregate members. If O contains subaggregates, this rule applies recursively to the subaggregates.

However, any of the non-outermost braces may be elided for aggregate initialization. In this case, the aggregate "consumes" only as many members of the Initializer as necessary to initialize the aggregate; the remaining members are left to continue initialization of the object of which the aggregate is a part.

The initialization construct changes the type of the O if O is of an incomplete type [?]:T. If V is the number of elements of O that are initialized in the Initializer, the type of O is changed to [V]:T.

When string constant E is used to initialize an object of an array type, the type of E is *not* converted to *C (where C is the type denoted by **char**) as is customary; see Sections *Expressions/<IDENTIFIER>s* and *Expressions/Member Selection.*

An array of characters may be initialized by a string. Successive characters of the string (including the terminating character if there is room or if no size is specified) initialize the members of the array. Likewise, a pointer to Signed-Char may be initialized to a string, since a string is always coerced into its address; see Section *Expressions/Constant Expressions.*

When initializing a object of a Union type, the first member of the union is initialized.

These rules are more precisely described in the following algorithm that formally establishes the correspondence between the initialized object and the initializing expressions. The algorithm is recursive and is written in "pseudo-C"; furthermore, it does not contain Initializers so that it non-circularly specifies the meaning of Initializers.

The algorithm takes as input an object V to be initialized and an Initializer I. The constraints that must be imposed

and the semantics of the initialization are marked in the algorithm. It is started with the call Initialize(V, I, 1).

```
int Initialize(V, I, Start)
   object V; Initializer I; int Start;
   {
   let I be of the form {E1, ..., En} or just E1,
      where the Ei are Expressions or Initializers;
   switch (the type T of V):
      case basic type: case *T':
         if EStart is of the form E or {E},
               where E is an expression,
            Constraint: The type of E must be assignment-
            compatible with the type of the object (permits assigning an array of characters into a *C object (where C is the type denoted by just "char"), since arrays are always converted to pointer to the first element, or the address of an object into a pointer).
            Semantics: V is initialized with the value of E.
            return Start+1;
         else
            Constraint: The initialization is erroneous:
                        too many braces for a scalar type.
      case [L]:T':
         if EStart is a string S of length L+1 or less
            (including the '\0' terminator)
         and T' is Signed-Char
            {/* Note: V+1 permits us to ignore the null */
             /* byte in S.                              */
            Semantics: V[i] is initialized with S[i],
                       for 0 ≤ i ≤ Min(L, length of S - 1);
            return Start+1;
            }
         else {
            for (k=Start, i=0; k ≤ Min(V-1, n) ; i++)
               if Ek is of the form {...}
               then {Initialize(V[i], Ek, 1); k++}
               else k = Initialize(V[i], I, k);
            return k;
            }
```

```
        case [?]:T': /* Initialization determines the size */
                                                /* of the array. */
            if Estart is a string S of length V
            and T' is the type denoted by "char" alone
                {
                Constraints: the type of V becomes [V+1]:T';
                Semantics:   V[i] is initialized with S[i],
                             for 0 ≤ i ≤ V;
                return Start+1;
                }
            else {
                for (k=Start, i=0; k ≤ n ; i++)
                    if   Ek is of the form {...}
                    then {Initialize(V[i], Ek, 1); k++}
                    else k = Initialize(V[i], I, k);
                Constraints: the type of V becomes [i+1]:T';
                return k;
                }
        case struct{Members}:
            let the members of the structure be numbered 1..M
            and denoted V [1]...V[M];
            for (k=Start, i=1; k ≤ Min(M, n)-1; i++)
                if   Ek is of the form {...}
                then {Initialize(V[i], Ek, 1); k++}
                else k = Initialize(V[I], I, k);
            return k;
        case union{Members}:
            let V[1] be the first member of the union;
            return Initialize(V[1], I, Start);
        /* case Parameters->T':   -- No case here,
            -- since functions cannot be initialized. */
    } }
```

When the algorithm ends, all of the Initializer expressions must have been "used". The value of Initialize must therefore be one more than the length of the Initializer list.

All uninitialized objects of static storage class are initialized to zero. Likewise, portions of static aggregates that are uninitialized by Initializers are initialized to zero. Uninitialized objects of storage class automatic, or uninitialized por-

tions of such objects in the case of aggregates, have undefined values. Initialization of automatic objects occurs when the declaration of the initialized object is elaborated; see Section *Statements/Compound_statement* for the definition of elaboration and when it occurs.

The evaluation of the expressions in an `Initializer` is performed from left-to-right. However, the order of initialization of the components of an aggregate is undefined. This means that constructs illustrated by "`struct {int x, y;} z = {1, z.x};`" are not well-defined.

Discussion

Both KR and X3J11 seem to allow initializing objects of storage class static-import. However, neither document ascribes any semantics to the initialization, especially in the situation where there are two distinct initializations for the same imported object. 4.2BSD uses ForTran-style named common as a means of sharing objects, so that it is up to the linker to choose which initialization specification among many prevails.

X3J11 has adopted the position that when an automatic aggregate is partially initialized by an `Initializer`, the remainder of the aggregate is initialized to zero. We consider this unfortunate and encouraging of poor programming style. In fact, we begrudgingly left in the initialization of all otherwise-uninitialized static objects to zero because of widespread existing practice. Depending upon this "free" initialization we feel is, again, poor programming practice.

Perhaps the reason the dependence is so widespread is that UNIX linkers zero-initialized otherwise-uninitialized memory automatically. Some operating systems do not do this and, rather than dedicate a lot of object module space to the zeroing of uninitialized space, we would prefer the language leave undefined the initial value of uninitialized objects, and require the programmer to be disciplined enough to initialized all and only those objects needing it. In this way he can document what objects need no initialization; with standard C one cannot tell.

As an example of an initialization, consider the Non_function_definition

```
struct {int x, y[2]; char c[2];} z[] = {
        1,      {2},    "a",
    {   1,      2,      3,      'x' },
        7
    }
```

The initialization specified is as follows:

```
z[0].x      = 1;
z[0].y[0]   = 2;
z[0].c      = "a";
z[1].x      = 1;
z[1].y[0]   = 2;
z[1].y[1]   = 3;
z[1].c[0]   = 'x';
z[2].x      = 7;
```

z[1].c[1] is left uninitialized and therefore is set to zero. The same is true of z[2].y and z[2].c. The type of the array z has been changed to [3]:Struct{...}.

```
Initialize(z, I, 1):
   case []:T', where T' is the struct type.
      k = Initialize(z[0], I, 1):
         case struct{...}:
            k = Initialize(z[0].x, I, 1):
               case basic type: z[0].x = 1;
               return 2;
            Initialize(z[0].y, {2}, 1):
               case [2]:int:
                  k = Initialize(z[0].y[0], {2}, 1):
                     case basic type: z[0].y[0] = 2;
                        return 2;
                  return 2;
            k = Initialize(z[0].c, I, 3):
               case [2]:char:
                  Since I3 is a string,
                     z[0].c[0] = 'a', z[0].c[1] = '\0';
                  return 4;
            return 4;
```

```
/* At this point k = 4. */
Initialize(z[1], {1, 2, 3, 'x'}, 1):
   case struct{...}:
      k = Initialize(z[1].x, {1,2,3,'x'}, 1):
         case basic type: z[1].x = 1;
            return 2;
      k = Initialize(z[1].y, {1,2,3,'x'}, 2):
         case [2]:int:
            k = Initialize(z[1].y[0], {1,2,3,'x'}, 2):
               case basic type: z[1].y[0] = 2;
                  return 3;
            k = Initialize(z[1].y[1], {1,2,3,'x'}, 3):
               case basic type: z[1].y[1] = 3;
                  return 4;
            return 4;
      k = Initialize(z[1].c, {1,2,3,'x'}, 4):
         case [2]:char:
            k = Initialize(z[1].c[0], {1,2,3,'x'}, 4):
               case basic type: z[1].c[0] = 'x';
                  return 5;
            return 5;
      return 5;
/* k is now 5. */
k = Initialize(z[2], I, 5):
   case struct{...}:
      k = Initialize(z[2].x, I, 5):
         case basic type: z[2].x = 7;
            return 6;
      return 6;
the type of z is [3]:T';
return 6;
```

Thus, the outermost call to Initialize returns six, one more than the number of elements in the Initializer, which is correct, and the type of z is [3]:Struct{...}.

```
char s[] = {'a', 'b', '\0'};
```

causes s's type to be [3]:Signed-Char and specifies its initialization it to the string "ab". This is equivalent to:

```
char s[] = "ab";
```

Note that

```
char *s = {'a', 'b', '\0'};
```

is *not* permitted, since s is not of an array type, but that

```
char *s = "ab";
```

has the same effect. However,

```
char s[2] = "ab";
```

sets s[0] = 'a' and s[1] = 'b'; the '\0' byte in "ab" is ignored. 4.2BSD and KR do not allow this case, an X3J11 extension.

The elision of {...}s is most useful in fully initializing arrays of arrays:

```
int Matrix[3][3] = { 1,2,3,  4,5,6,  7,8,9};
```

However, an upper triangular Matrix can be specified by using {...}s to partially initialize the rows of the Matrix:

```
int Matrix[3][3] = {  1,2,3,  {5,6},  {9}};
```

This produces the matrix

```
1  2  3
5  6  0
9  0  0
```

since uninitialized storage is set to zero.

The initialization of structure objects with a structure-valued expression is an X3J11 extension.

Different strategies. In existing compilers there are two strategies for initialization in the presence of the elision of braces — i.e. when the Initializer is not fully "braced". This technique, which we call *top-down*, differs from that of 4.2BSD, which we call *bottom-up*.

In the top-down approach, missing braces are interpreted as missing from the most nested components of an object being initialized. Put another way, the structure of the braces in the Initializer matches the structure of the object being ini-

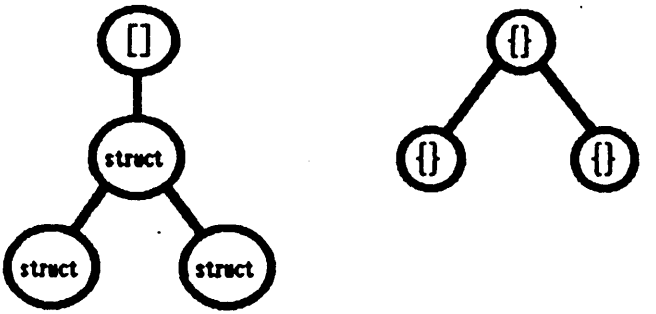
tialized from the top. If there are missing braces, only the top portion of the structure of the initialized object is matched.

In the bottom-up approach, missing braces are interpreted as missing from the least nested components of an object being initialized.

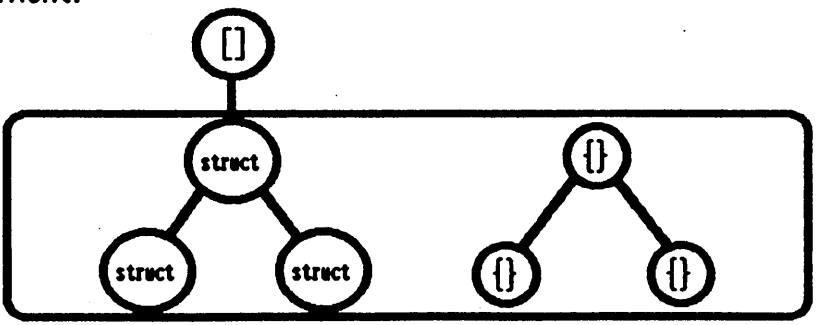
The following example illustrates the difference:

```
struct { struct {int x;} a, b; } z[] = { {1}, {2} };
```

The trees below depict the structures of the initialized object (left): and the Initializer (right):

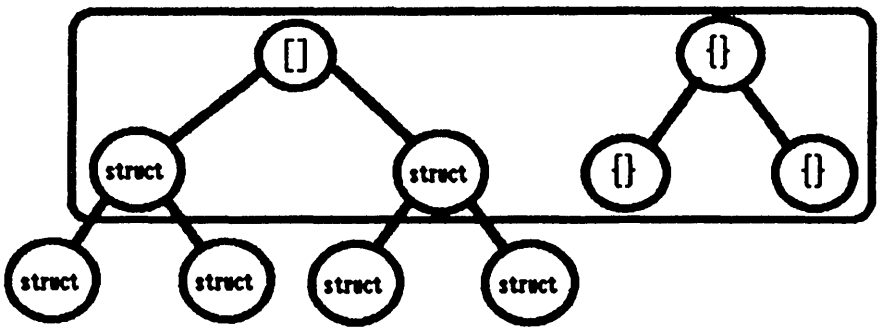


In the bottom-up approach, the Initializer's structure matches the bottom portion of the tree. The initialization is z[0].a.x = 1 and z[0].b.x = 1, so that z is an array of one element:



In the top-down approach, the Initializer's structure matches the top portion of the tree, forcing a bifurcation of

the top node so that z has two elements. The initialization is z[0].a.x = 1 and z[1].a.x = 2:



High C supports the top-down approach because it seems the only approach a human can easily take. All compilers that we know of, except 4.2BSD, use the top-down approach. The bottom-up approach may perhaps be an artifact of the compilation strategy used in 4.2BSD.

7

Statements

A *statement* specifies an action to be performed.

7.1 Compound Statement

[pragma; elaboration of declarations; intermixing declarations and statements]

Syntax

```
Compound_statement
-> '{' (Specified_declaration|Pragma_call|Statement)* '}'
;
Statement
-> Compound_statement
```

Constraints

When a Compound_statement is not the body of a Function_definition, the Compound_statement itself forms a block. Otherwise, a block is formed that includes the Parameters and Parameter_types of the Function_definition; see Section *Declarations/Function Definitions*.

Semantics

Except for the **goto**, **break**, and **continue** Statements, the declarations and statements of a Compound_statement are executed in the same sequence as their textual presentation.

The "execution" of a declaration is a term perhaps unfamiliar to the reader. In Ada it is instead called *elaboration*, and we shall also use this term. The elaboration of a declaration consists of the initialization of the object, if an initializer is present and the object's storage class is automatic.

As we have mentioned in Section *Concepts/Lifetimes*, storage for an automatic object is allocated at any entry to the Compound_statement containing the object's definition, so

that elaboration does not include storage allocation.

Discussion

No other C language definition we know of permits the intermixing of declarations and statements. This flexibility allows the placement of declarations closer to the uses of the declared object. ***Example:***

```
getrand(int a[100]) {
   int i;
   for (i = 1; i < 100; a[i++] = rand());
   }

main() {
   int i, j, a[100];
   for (j = 1; j <= 10; j++) {
      getrand(a);
      int max = 0; /* Declaration after statement. */
      for (i = 1; i < 100; i++)
         if (a[i] > max) max = a[i];
      printf("Maximum of random values=%d\n",max);
   } }
```

Although one might argue that the intermixing buys little convenience, we approached it from the other point of view and saw no advantage gained by the restriction, so we did not impose it. Furthermore, the approach yields a simpler semantics for the timing of the initialization of objects declared with initializers, and when such initialization is by-passed (via **goto** or other such jump).

The semantics of local lifetime (see Section *Concepts/Lifetimes*) together with the scope rules allow automatic objects in parallel Compound_statements to overlap on a run-time stack for storage efficiency.

7.2 Expressions as Statements

[side effects]

Syntax

```
Statement
-> EL ';'
```

Constraints

None.

Semantics

The expression list EL is evaluated and the resulting value discarded.

Discussion

The primary purpose of an expression as a statement is to achieve a side effect, such as via a function call or an assignment.

7.3 switch, case, and default

[sequence point; case ranges; initialization; elaboration of declarations]

Syntax

```
Statement
-> 'switch' '(' EL ')' Switch_body:Statement
-> 'case' Case_label:(Constant:E ('..' Constant:E)?)
            ':' Statement
-> 'default' ':' Statement
```

Constraints

The EL of the switch statement must have integral type. Consider the set of Statements S that appear within the Switch_body but do not appear within any nested Switch_body. Let SC denote the subset of S whose members are labeled with **case** and SD the subset labeled with **default**.

In the E..E form of a Case_label, both Es must be of the same integral type and must be constant expressions such that

the value of the first is no greater than that of the second; call the values bounded by the two values the *range* of the Case_label. Where this form of Case_label appears in SC, the type of the two Es must be assignment-compatible with th type of EL.

Where the Case_label of a Statement in SC is a single E, E's type must be assignment-compatible with that of EL. Furthermore E must be a constant expression. The range of this Case_label is the single value denoted by E.

There may not exist two Statements in SC having overlapping Case_label ranges. The size of SD must be zero or one, i.e. there may be at most one **default** within a **switch**.

case or **default** Statements may appear only within **switch** Statements.

Semantics

When the **switch** Statement is executed, the value V of EL is computed, and the end of its evaluation is a sequence point. I V is contained in the range of the Case_label of some Statement S in SC, control is transferred to S. If no Statement in SC has a Case_label whose range contains V, and SD contains a Statement D, control is transferred to D. If no Case_label's range contains V and SD is empty, the **switch** Statement has no further effect.

> *Note.* When control is transferred to some Statement S, no initialization of any objects of storage class automatic declared in Switch_body is performed prior to the control transfer. This is because the declarations are not elaborated, having been "skipped over" by the control transfer.

The execution of a **case** or **default** Statement has the same semantics as executing the Statement labeled by these constructs. Thus a transfer to such a Statement from elsewher within a **switch** Statement (via, e.g., a **goto**, or by "flowing off the end" of one case onto the next) is meaningful.

Discussion

The first example shows a "normal" use of the **switch** construct:

```
switch (Char_class(c)) {
   case Letter:          /* Scan identifier. */
      while ((c = getc()) >= 'a' && c <= 'z');
      break;
   case Number:          /* Scan number. */
      while ((c = getc()) >= '0' && c <= '9');
      break;
   case Space:           /* Scan blanks. */
      while ((c = getc()) = ' ');
      break;
   default:
      printf("Illegal token.\n");
   }
```

This next example points out the ill-disciplined potential of the **switch** construct and alerts unsuspecting implementors to the construct's full "power":

```
switch (i) default: {
   case 0:
   if (1) case 1: printf("case 0\n");
   else   case 2: printf("case 1\n");
   /* This loop cannot be "optimized away"  */
   /* because it may be entered via switch. */
   while (0) {
      int j = 2*i;
      /* Initialization will never be performed. */
      case 3: printf("Entered a while false loop; \
                      j is garbage:%d\n", j);
   } }
case 55..77:
default:     printf("Execution of case finished.\n");
```

The last statement is executed when the first **switch** terminates; the Case_labels 55..77 and **default** are irrelevant (and illegal) unless the example above appears in the context of another **switch** Statement.

7.4 if

[sequence point]

Syntax

Statement
-> 'if' '(' EL ')' Statement ('else' Statement)?

The grammar as it stands is ambiguous. The ambiguity is resolved as follows: in an **if** Statement with an **else** phrase, the Statement following the (EL) may not end in an **if** Statement that lacks an **else**. Said differently, the **else** is matched with nearest lexically preceding **if** without an **else** in the same Compound_statement (but not in a nested Compound_statement). Thus in

```
if (1) while (1) if (1) {} else {}
```

the **else** matches the second **if**.

Constraints

EL must have scalar type.

Semantics

EL is evaluated; the end of its evaluation is a sequence point. If it is non-zero, the first Statement is executed. If it is zero, the second Statement, if supplied, is executed.

If control transfers to the first Statement or any Statement contained within it by any other means (such as with a **goto** or **switch**), the second Statement, if supplied, is not executed, and upon termination of the first Statement, control flows to the Statement after the **if** Statement. Likewise with control transfers to the second Statement: the first Statement is not executed.

7.5 while

[sequence point]

Syntax

```
Statement
-> 'while' '(' EL ')' Statement
```

Constraints

EL must have scalar type.

Semantics

EL is evaluated; the end of its evaluation is a sequence point. If the value is non-zero, the contained Statement is executed. If the contained Statement S does not transfer control outside of the **while** Statement, then upon S's termination the **while** Statement is executed again.

7.6 do-while

[sequence point]

Syntax

```
Statement
-> 'do' Statement 'while' '(' EL ')' ';'
```

Constraints

EL must have scalar type.

Semantics

The contained Statement is executed. If the contained Statement S does not transfer control outside of the **do** Statement, then upon S's termination EL is evaluated; the end of its evaluation is a sequence point. If the value is non-zero, the **do** Statement is executed again.

7.7 for

[sequence point; continue]

Syntax

```
Statement
-> 'for' '(' First:EL?
         ';' Next: EL?
         ';' Last: EL?
         ')' Body: Statement
```

Constraints

Next, if present, must have a scalar type.

Semantics

If Next is omitted, one is implied.

Except in the matter of the behavior of a contained **continue** Statement in the Body, this statement is exactly equivalent to

```
First; while (Next) { Body; Last; }
```

Thus, the First and Next evaluation ends are sequence points.

Discussion

The first expression is a convenient place to put any initialization for the loop. The second specifies a test for continuing the loop and perhaps an incrementation. The third usually specifies an operation performed at the end of the loop, such as an increment of a variable.

7.8 gotos and Labels

[initialization; scope of labels; elaboration of declarations]

Syntax

```
Statement
-> 'goto' Target_label:Name ';'
-> Labeled_statement: Label:Name ':' Statement
```

Constraints

The beginning of the Compound_statement of the function containing the Label:Name is the Name's defining point; the label is declared in the label name space. The scope of the label is the block associated with that function. Note that this is an exception to the normal rule that scope extends from the defining point to the end of a Name's origin. There may not be two identical Label:Names in the same function — labels can not be re-declared in a nested Compound_statement.

The Target_label of a **goto** Statement must be declared in the label name space.

Semantics

A **goto** Statement transfers control to the Statement prefixed by the associated Target_label.

The semantics of a Labeled_statement is the same as that of the contained Statement.

Note. When a Compound_statement S is entered via a **goto**, no initialization of objects of storage class automatic declared within S is performed. This is because such initialization occurs only when the declarations are elaborated, and the **goto** "skips" the elaboration.

Discussion

The scope of label names is different from that described in KR, for good reasons. Consider the function

```
int f() {
   L1: ;
      { goto L1;      /* Target is the last L1. */
        {L1: ; }
        L1: ;
   } }
```

KR seems to say that the scope of the innermost L1 is the entire function ("The scope of a label is the current function, excluding any sub-blocks in which the same identifier has been

redeclared." [K&R, p.204]). High C forbids re-declaration of a label within a function.

High C constraints agree with those of X3J11 and 4.2BSD. This makes most sense since it avoids the pitfalls of the above example where a reader could assume the **goto**'s target is the first L1 instead of the third L1. If a lot of program text separated the **goto** and the third L1, the target would be difficult to see.

7.9 break

[exiting a switch, for, while, or do statement]

Syntax

```
Statement
-> 'break' ';'
```

Constraints

The **break** Statement must be contained within a **switch**, **for**, **while**, or **do** Statement.

Semantics

The **break** Statement terminates the execution of the smallest enclosing **switch**, **for**, **while**, or **do** Statement.

7.10 continue

[continuing a for, while, or do statement]

Syntax

```
Statement
-> 'continue' ';'
```

Constraints

The **continue** Statement must be contained within a **for**, **while**, or **do** Statement.

Semantics

The **continue** Statement jumps to the loop-continuation portion of the smallest enclosing **for**, **while**, or **do** Statement. More precisely, in each of the Statements

```
while (...)   {Body: Statement; Continue.;};
do            {Body: Statement; Continue.;} while(...);
for   (...)   {Body: Statement; Continue.;}
```

a **continue** contained within the Body Statement and not contained within any contained **for**, **while**, or **do** Statement is equivalent to a **goto** *Continue*.

7.11 return

[result of function call]

Syntax

```
Statement
-> 'return' EL? ';'
```

Constraints

The **return** Statement must be contained within a function F. The type of EL must be assignment-compatible with F's return type.

Semantics

The **return** Statement causes termination of the currently executing function and returns control to its caller. If the EL is present, it is evaluated, converted to the return type of the function in which it appears, as if by assignment, and this value is returned to the caller. If the caller expects no value, the behavior is undefined.

If EL is omitted, no value is returned. If a value is expected by the caller, the behavior is undefined.

An implicit **return** (with no EL) is assumed at the end of every function.

Discussion

In KR, all functions had return types, so values could always be returned. Returning without a value returned an undefined value. In our semantics, which agree with X3J11, potentially worse behavior is permitted, such as abnormal program termination.

7.12 The Null Statement

Syntax

```
Statement
-> ';'
```

Constraints

None.

Semantics

None. This statement is most often used to carry a label.

8

Expressions

8.1 General

[precedence, associativity in expressions; evaluation order; commutativity, associativity; expression rewriting; sequence point; arithmetic conversions; Convert; extended floating-point precision; *, +, &, ^, and | commutative and associative]

Expressions are presented with full operator precedence and associativity rules contained unambiguously in the grammar. Consequently there are many "chain" productions of the form "E_n -> E_{n+1}". Often, the grammar is listed in the form

E_n -> E_{n+1} -> Interesting_alternative

where E_n -> E_{n+1} is the chain production, and Interesting_alternative is actually the material to be discussed. It is always true that the constraints and semantics of E_{n+1} are the same as those of E_n. Therefore all references in the text refer to nonterminals or adjectives present in Interesting_alternative, and never refer to E_{n+1}.

The order of evaluation of operands is not defined unless specifically stated otherwise — for the (), &&, ||, ?:, and comma operator. However, the evaluation of operands of an operator must not be "interleaved": one operand of an operator must be completely evaluated before the evaluation of another operand commences. This essentially forces a "top-down" bent to evaluation of an expression.

The operators *, +, &, ^, and | are commutative and associative and a language processor is free to rewrite any expression involving these operators using the commutativity and asociativity rewrite rules, provided that the types of the operands or of the results are not changed in the process. However, once rewriting has been done, operand evaluation order must be as described in the previous paragraph, so that if "$(e_1 + e_2) + e_3$" is rewritten as "$e_1 + (e_2 + e_3)$", the outer

addition requires that e_1 be completely evaluated before (e_2 + e_3) is, or vice-versa.

Formally, the rewriting rules and constraints on using them are as follows:

- commutativity: e_1 **op** e_2 <=> e_2 **op** e_1 always.
- associativity: for **op** in {*, +, &, ^, |},

$$(e_1 \textbf{ op } e_2) \textbf{ op } e_3 \Leftrightarrow e_1 \textbf{ op } (e_2 \textbf{ op } e_3)$$

$$\text{iff} \quad \text{Type}(e_1 \textbf{ op } e_2) = \text{Type}(e_2 \textbf{ op } e_3). \qquad \text{(C)}$$

Since the first (e_1) and second (e_3) operands of the first and second operator occurrences, respectively, are the same, the condition guarantees that second and first operands of the first and second operator occurrences, respectively, have the same type, so that the types of the operands and results of both operator occurrences are preserved by the transformation.

That the single condition (C) guarantees operand-type sameness cannot be evident without knowing the conversion rules that are applied to the operands of the operators before the operation commences. Consider the first operator.

> What is the type of its second operand on the left side of the <=>? Before the operation commences, both operands are converted to type Common(e_1,e_2), which is therefore the actual type of *both* operands, and is the type Type(e_1 **op** e_2) of the result.
>
> Now consider the second operand on the right side of the <=>. Its type is evidently just Type(e_2 **op** e_3).
>
> Therefore we arrive at the requirement that Type(e_1 **op** e_2) = Type(e_2 **op** e_3), which is just (C).

An analysis of the first operand of the second operator produces the identical requirement.

This covers half the cases. We now consider the first operand of the second operator, and the second operand of the first operator.

The first operand of the second operator, on the right of <=>, is of type Type(e_1 **op** (e_2 **op** e_3)). But by (C), this is the same as Type(e_1 **op** (e_1 **op** e_2)). This now reduces to Type(e_1 **op** e_2), since Common(e_1,Common(e_1,e_2)) = Common(e_1,e_2); see the definition of Common in Section *Concepts*. Now the type of the first operand of the second operator, on the left of the <=>, is evidently already Type(e_1 **op** e_2).

Therefore (C) guarantees that the type of the first operand of the second operator is the same on both sides of the <=>. Similar arguments show that the type of the second operand of the second operator is the same on both sides of the <=>. Hence (C) is the only requirement necessary.

For example, as a consequence of (C), (i+j)+k can be rewritten as i+(j+k) if i,j, and k are all of type Unsigned-Int, but cannot if k is of type Double or Float.

The only sure way of placing an order upon evaluation and any involved side-effects is to introduce a sequence point.

Many operators make use of arithmetic conversions. The conversions are detailed in Section *Concepts*. Functions Common and Widen are used to describe types to which operands are converted. In the sequel, when we specify that a value V is converted to type T by writing Convert(V,T), we mean V if V is of type T, and the result of converting V to T by assignment-compatibility rules if V is not of type T.

When operands of an operator are converted to a floating-point type T, and the result of the operation is type T, we permit an implementation to choose a different floating point type T' having no less precision and range than T that the operands and result may be represented in. As far as the constraints of the expression are concerned, the type of the result remains T, but the implementation may store that result in T', preserving more precision. The type T' need not even be available to the C programmer — e.g. it may even be of greater range and precision than Long-Double. The implementation must provide for casting the (invisible) type T' to T when the context demands, as in an assignment.

This permission is important on architectures that have a natural most-extended precision and range floating point type in which all computation is normally done, such as for the Intel 8087 chip, and has its main impact upon intermediate expressions. *Example:*

```
double d1, d2, d3, d4, d5;
...
d1 = (d3*d4)-(d2*d1);
    /* Here, Long-Double may be used for calculation, possibly     */
    /* obtaining more precision than normal, or avoiding an error */
    /* if d3*d4 or d2*d1 exceeds the range of Double.              */
    /* The Long-Double result must be cast to Double before       */
    /* the assignment takes place.                                */
d5 = (d3*d4)-(d1+d2);
    /* Furthermore, the common subexpression d3*d4 may be stored  */
    /* in a maximally precise temporary format for use here.      */
```

All expressions have a type T. The result of an operation denoted by an expression is defined only if the computed value is of type T, with the exception just noted where an implementation may choose a "higher" floating-point type T'. In addition, if any operand of an operation is undefined, the value of the result is undefined.

8.2 Comma Operator: , ----------

[sequence point; forcing evaluation order]

Syntax

EL -> E ',' EL_2
   -> E;

Constraints

The type and mode of the expression list EL is the type and mode of EL_2 (the subscript 2 is used here for convenience of reference and is not part of the formal grammar).

Semantics

E is evaluated, then EL_2; the value of EL is the value of EL_2. The end of the evaluation of E is a sequence point.

8.3 Assignments: = ----------•

[right part of assignment; assignment compatibility]

Syntax

```
E   -> E1;
E1  -> E2
    -> Lvalue:Term' Plain_assignment:'=' E1
    -> Lvalue:Term' (Assignment_operator E1);
Assignment_operator
-> '|=' | '^=' | '&=' | '>>=' | '<<='
 | '+=' | '-=' | '*=' | '/='  | '%=';
```

Constraints

The type T of an Assignment_expression is the type of its Lvalue. The mode of the Lvalue must be **var** or **field**.

Define the *right part* of the assignment operator as E1 if Plain_assignment is used, and "Term' **op** (E1)" if Assignment_operator is used and is of the form **op**= (where **op** is |, ^, &, etc.). In the latter case "Term' **op** (E1)" must satisfy the constraints for **op** — discussed below separately for each **op**.

The type of the right part must be assignment-compatible with T, or be a constant integral expression evaluating to 0 and T of a pointer type. The mode of the result is **value**.

Semantics

The value V of the right part and the variable L referred to by the Lvalue are determined, in an unspecified order. V' = Convert(V,T) replaces the value held by L. Where Assignment_operator is used, the Lvalue must be evaluated only once.

The value of the Assignment_expression is V'.

If V is obtained from an object that overlaps in storage with L, the semantics of assignment is undefined.

Discussion

Note that the requirement on the mode of the Lvalue prevents assignments into a name of mode **typedef** or **fcn**. A structure member as the left side of an assignment is generally of mode **var**, not **member** (see Subsection *Member Selection* below), so that assignments into structure members are permitted. Objects of mode **tag** or **label** are never encountered since they exist only in name spaces from which expressions cannot come.

8.4 Conditional Expressions: ? : ----------•

[sequence point]

Syntax

```
E2  -> E3
    -> Conditional_expression:
       E3 '?' EL ':' E2
```

Constraints

The Conditional_expression's first operand E3 must be of a scalar type. The type T of the entire Conditional_expression is determined from the types T_{EL} and T_{E2} of its second and third operands EL and E2. T_{EL} and T_{E2} must be either both arithmetic, in which case T is Common(T_{EL},T_{E2}); or the two must be compatible types, in which case T is either T_{EL} or T_{E2} (it does not matter which); or one must be of a pointer type P and the other a constant expression evaluating to zero, in which case T is P. The mode of the result is **value**.

Semantics

The first operand E3 is evaluated; the end of its evaluation is a sequence point. If it is non-zero, the value V of the second operand EL is determined; otherwise, the value V of the third operand E2 is determined. (Therefore, only one of E1 and E2 is evaluated.) Convert(V,T) is the result.

8.5 Sequential Disjunction: || ----------•

[sequence point]

Syntax

```
E3  -> E4    -> E3 '||' E4;
```

Constraints

Each operand E3 and E4 of the || expression must have scalar type. The type of the || expression is Signed-Int. The mode of the result is **value**.

Semantics

Expression E3 is evaluated; the end of its evaluation is a sequence point. If it is non-zero, the result is one. If it is zero, expression E4 is evaluated. If it is non-zero, the result is one; otherwise the result is zero.

8.6 Sequential Conjunction: && ----------•

[sequence point]

Syntax

```
E4  -> E5    -> E4 '&&' E5;
```

Constraints

Each operand E4 and E5 of the && expression must have scalar type. The type of the && expression is Signed-Int. The mode of the result is **value**.

Semantics

Expression E4 is evaluated; the end of its evaluation is a sequence point. If it is zero, the result is zero; if non-zero, expression E5 is evaluated. If it is zero, the result is zero; otherwise the result is one.

8.7 Bit-wise Inclusive-or: | ----------•

Syntax

```
E5  -> E6   -> E5 '|' E6;
```

Constraints

Each operand E5 and E6 must have integral type. The type T of the | expression is Common(type of E5, type of E6). The mode of the result is **value**.

Semantics

Both operands E5 and E6 are evaluated and converted to type T. The bit-wise inclusive-or of the two operands is the result.

8.8 Bit-wise Exclusive-or: ^ ----------•

Syntax

```
E6  -> E7   -> E6 '^' E7;
```

Constraints

Each operand E6 and E7 must have integral type. The type T of the ^ expression is Common(type of E6, type of E7). The mode of the result is **value**.

Semantics

Both operands E6 and E7 are evaluated and converted to type T. The bit-wise exclusive-or of the two operands is the result.

8.9 Bit-wise And: & ----------•

Syntax

```
E7  -> E8    -> E7 '&' E8;
```

Constraints

Both operands E7 and E8 must have integral type. The type T of the & expression is Common(type of E6, type of E8). The mode of the result is **value**.

Semantics

Both operands E7 and E8 are evaluated and converted to type T. The bit-wise and of the two operands is the result.

8.10 Equality Comparisons: == and != ---------- •

Syntax

```
E8  -> E9    -> E8 '==' E9
             -> E8 '!=' E9;
```

Constraints

Let T_8 and T_9 be the types of E8 and E9. Both T_8 and T_9 must be arithmetic types, or be of compatible pointer types *T', or one must a constant integral expression evaluating to zero and the other a pointer type. The type T of the entire expression is Signed-Int. The mode of the result is **value**.

Semantics

Both operands E8 and E9 are evaluated. If T_8 and T_9 are both arithmetic types, then the values are converted to type Common(T_8,T_9). The (possibly converted) values are compared for equality (==) (inequality (!=)). The result is one in the case the values are equal (unequal); otherwise, the result is zero.

8.11 Ordering Comparisons: < > <= >= -------- •

Syntax

```
E9  -> E10  -> E9 '<'  E10
            -> E9 '>'  E10
            -> E9 '<=' E10
            -> E9 '>=' E10;
```

Constraints

The same as for the equality comparisons just described.

Semantics

Both operands are evaluated. If the types of both are scalar, the values are converted to the Common type of the two types. The (possibly converted) values are compared according to the specified relation: < for less-than, > for greater-than, <= for less-than-or-equal-to, and >= for greater-than-or-equal-to. The result is one if the relation is true and zero if false.

The comparison of two pointers is done as if they were unsigned integers of the appropriate length. The result is guaranteed only for two pointers that point into the same aggregate; otherwise the result is implementation-defined.

8.12 Shift Operators: << and >> ---------- •

Syntax

```
E10 -> E11  -> E10 '>>'E11
            -> E10 '<<'E11;
```

Constraints

Both operands must be of integral type. The type T of the result is Widen(type of E10). The mode of the result is **value**.

Semantics

The values V_{10} and V_{11} of E10 and E11 are determined. V_{10} is converted to T and V_{11} to Signed-Int. The value of E10 << E11 is V_{10}, interpreted as a bit pattern, left-shifted V_{11} bits, with this result interpreted as type T. The value of E10 >> E11 is V_{10}, interpreted as a bit pattern, right-shifted V_{11} bits, with this result interpreted as type T. The right shift is guaranteed to be logical (zero-filled) if E10 has an unsigned type; otherwise it may be arithmetic (filled with a copy of the leftmost bit). The result is implementation-defined if V_{11} is negative or greater than or equal to the size in bits of V_{10}.

8.13 Additive Operators: + and - ---------- •

[pointer arithmetic; addition, subtraction]

Syntax

```
E11 -> E12  -> E11 '+' E12
            -> E11 '-' E12;
```

Constraints

Let T_{11} be the type of E11 and T_{12} the type of E12, and let R be the type of the result. One of the following conditions must obtain:

(a) T_{11} and T_{12} are both arithmetic types, in which case R is Common(T_{11},T_{12});

(b) one is of the form *T and the other an integral type I, in which case R is *T (and if the operator is '-', the first operand must be the one of the form *T); or

(c) or T_{11} and T_{12} are the same pointer type *T and the operator is '-', in which case R is either Signed-Short-Int, Signed-Int, or Signed-Long-Int (which one in particular is implementation-defined).

In cases (b) and (c), in *T, T must not be a functionality type. The mode of the result is **value**.

Semantics

Both operands are evaluated.

In case (a), the values are converted to R, and the result is the sum ('+') or difference ('-') of the operands.

In case (b), if the value V of type *T points to the i^{th} element of an array of type [...]:T, the result is a pointer to element i+(value of type I) or i-(value of type I) of the array. This holds only for pointers within the bounds of the array, except that it is allowed to point to a hypothetical element following the array's last element. The use of the pointer result as an operand of * is defined only for pointer values within the array bounds.

In case (c), if the two values point to the i^{th} and j^{th} elements, respectively, of the same array, or possibly one past the end of the array, the result is i-j; otherwise the result is undefined. i-j must be a value of type R.

Discussion

The semantics of pointer arithmetic when the result exceeds the array bounds by one is necessary so that common C idiom "A[V]" for A of type [V]:T is reasonable; this semantics was proposed by X3J11. KR does not treat the matter. The 4.2BSD implementation agrees with our semantics.

8.14 Multiplicative Operators: * / % ---------- •

[multiplication, division, modulo]

Syntax

```
E12 -> E13  -> E12 '*' E13
            -> E12 '/' E13
            -> E12 '%' E13;
```

Constraints

Let T_{12} be the type of E12 and T_{13} the type of E13. Both operands must be of arithmetic type; the type T of the result

is Common(T_{12},T_{13}). For the operator %, each operand must be of integral type. The mode of the result is **value**.

Semantics

Both operands are evaluated and converted to type T.

If the operator is *, the result is the product of the two values.

If the operator is /, the result is the quotient. If the operands are both of integral type and the result of the division is not an integer, the result is as follows: if both operands are positive, the result is the largest integer less than the true quotient. If either operand is negative, the result is either the greatest integer contained in or the least integer containing the true result; which one in particular is implementation-defined.

The result of the operator % for values a and b is a - (a/b)*b, where / is the division operator explained above.

3.15 Type Casts

Syntax

```
E13 -> Term;
Term
-> Term'
-> '(' Cast_type ')' Term
;
```

Constraints

The Cast_type must be of a scalar type C or type Void, and the term must be of some scalar type T, unless casting to Void. The type of the result is C. The mode of the result is **value**.

Semantics

The value V of the Term is determined.

If T is a pointer type and C some integral type, the result is undefined if an object of type C cannot hold V. Otherwise the result is V.

If T is some integral type and C a pointer type, the result is undefined if an object of type C cannot hold V. Otherwise the result is V.

If T and C are both pointer types to functions, the result is V.

If T and C are both pointer types to non-function objects, the result may not be defined if the alignment for C is more restrictive than that of T. Otherwise, the result is V.

Otherwise, the result is Convert(V,T).

Discussion

The semantics of pointer conversion requires that pointers to non-function objects be the same size.

Subset `Term'` of `Term` has been introduced to capture the relative precedences of the cast syntax and of **`sizeof`**: **`sizeof`** binds more tightly than a cast. Therefore "**`sizeof(int)*x`**" is the same as "**`(sizeof(int))*x`**".

Casting to Void is most often used to discard the result of a function, when writing an expression that is a function call.

8.16 Pointer Dereference: * ---------- •

[pointer alignment]

Syntax

```
Term'
-> T1
-> <'*'> Pointer:Term
```

Constraints

The Pointer must be of a type of the form *T. The type of the result is T, except where T is of the form []:T', in which case the type of the result is *T'. The mode of the result is

var unless T is a functionality type, in which case the mode is **value**. T must be neither an incomplete type nor Void.

Semantics

The value V of the Pointer is determined. The result is the object pointed to by V. If V is zero the operation is undefined.

Through type casts it is possible to obtain a pointer value V that is inappropriately aligned for the pointed-to type T. V is therefore not a valid value of type *T and therefore the operation *V is undefined.

Discussion

4.2BSD allows *ⁿf, for n ≥ 1, to denote *f for a value f of functionality type. High C prohibits such, as do KR and X3J11.

8.17 Pointer Reference: & ----------

[conversion of arrays to pointers; address of an array]

Syntax

```
Term'
-> '&' Term
```

Constraints

The Term may be of any type T. The result is of type *T. The Term must be of mode **var**, and may not denote an object declared with Storage_class **register**. The mode of the result is **value**.

Note. If Term is of an array type, the normal conversion of its type to *T is not done; see Subsections *<IDENTIFIER>s* and *Member Selection* below.

Semantics

The result is a pointer to the Term.

Discussion

Note that taking the address of a structure field is prevented because fields are of mode **field**.

The constraints prohibit &F, where F is a function name, since by the function conversion rules (see Subsection *<IDENTIFIER>s* below), F is converted to the address of F of mode **value**. Most compilers simply warn when &F is used; 4.2BSD permits &^{n}V, for n ≥ 1, to denote &V, but High C disallows such.

X3J11 introduced the ability to take the address of an array expression, where in KR the automatic conversion of array expressions to pointers to the first element made this impossible. The added ability allows the construction of an pointer of type *[]:T from an (array) expression of type []:T. Previously it was only possible to obtain type []:*T, and *[]:T had to be obtained through casting. ***Examples:***

```
int a[10];      /* Type []:Signed-Int.                          */
void f(int (*arg)[]) {...}
                /* Parameter is of type *[]:Signed-Int.         */
f(a);           /* Illegal: argument is of type *Signed-Int.    */
f((int(*)[])a); /* Old, tedious way of getting around problem. */
f(&a);          /* New: obviates need for cast.                 */

typedef t[10];
t a;            /* Declare an array.                            */
t *b = &a;      /* Initialize b to the array's address.         */
t *c = (t *)a;  /* Equivalent, old method.                      */
```

8.18 Unary Sign Operators: - and + ---------- •

[negation]

Syntax

```
Term'
-> '-' Term
-> '+' Term
```

Constraints and Semantics

Same as those of (0-Term) and (0+Term), respectively.

Discussion

Note that "+x" is not necessarily the same type as x; if x is of type Signed-Char, "+x" is of type Signed-Int.

8.19 Bit-wise Complement: ~ ---------- •

Syntax

```
Term'
-> '~' Term
```

Constraints

The Term must be of an integral type T. The type of the result is Widen(T). The mode of the result is **value**.

Semantics

The Term is evaluated and converted to type Widen(T). The result is the bit-wise complement of the value.

8.20 Boolean Negation: ! ---------- •

Syntax

```
Term'
-> '!' Term
```

Constraints and Semantics

Same as for the expression (0 == Term).

8.21 sizeof

[conversion of arrays to pointers; byte]

Syntax

```
Term'
-> 'sizeof' ('(' Cast_type ')' | Term')
```

Constraints

Term' may not be of mode **field**. Let T be the Cast_type, or the type of the Term'. T must be neither Void nor any functionality type nor any incomplete type. The result is of an integral type; which one in particular is implementation-defined. The mode of the result is **value**.

Note. If Term' is of an array type, the normal conversion of its type to *T is not done — see Subsections *<IDENTIFIER>s* and *Member Selection* below.

Semantics

Where Term' is used, it is *not* evaluated. In any case th result is a non-negative integer that is the number of bytes required to hold a value of type T. It is true that **sizeof**([V]:T) = V***sizeof**(T) and that **sizeof**(Signed-Char) = **sizeof**(Unsigned-Char) = 1.

Discussion

The reason that the automatic conversion of array types to pointers is avoided is so that it is possible to obtain the size of an array, rather than the size of a pointer to the array's element type. *Example:*

```
char a[10];
int j = sizeof(a);    /* j is initialized with 10.          */
void f(char b[]) {
   int k = sizeof(b);
   /* k is initialized with the size of a pointer to char, since *
   /* a's declaration is adjusted to read "char *a;". The same  */,
   /* initial value would be supplied for j above if we did not  */
   /* defeat the normal conversion of array types.               */
   }
```


8.22 Prefix Increment and Decrement: ++ and -- -- •

[addition, subtraction]

Syntax

```
Term'
-> T1
-> '++' Lvalue:Term
-> '--' Lvalue:Term;
```

Constraints

The Lvalue must be of mode **var** or **field** and scalar type T. The result is of type T and mode **value**.

Semantics

The Lvalue is evaluated. The value held by the object referenced by the Lvalue is incremented (++) (decremented (--)), i.e. one is added (subtracted), with the semantics of addition (subtraction) described above. The result is the incremented (decremented) value.

8.23 Postfix Increment and Decrement: ++ and -- - •

[addition, subtraction; side-effect, sequence point]

Syntax

```
T1
-> Lvalue:T1 '++'
-> Lvalue:T1 '--';
```

Constraints

The Lvalue must be of mode **var** or **field** and scalar type T. The result is of type T and mode **value**.

Semantics

The Lvalue is evaluated. The result is this value. The value held by the object referenced by the Lvalue is incremented (++) (decremented (--)), i.e. one is added (subtracted), with the semantics for addition (subtraction) described above.

The incrementing (decrementing) operation is a side-effect so it can be postponed until the next sequence point.

8.24 Function Call: () ----------

[functionality type and prototype functionalities; argument type checking; variable number of arguments to a function; recursive functions; Pascal function call semantics; sequence point]

Syntax

```
T1
-> T1 '(' Arguments ')';
Arguments
-> (E list ',')?;
```

Constraints

If T1 is a Name that is undeclared in the ordinary name space, the occurrence of the Name becomes its defining point in the ordinary name space as a function of type F = (?) -> Signed-Int with storage class static-import and mode **fcn**. If so, this replaces the normal constraints for the T1, which require that the Name is declared. The scope for this Name extends from its defining point to the end of the program, which differs from normal block-structured scope rules.

Otherwise, T1 must be of some functionality type F.

F is one of the following four forms; see Section *Concepts:*

(a) (?)	-> T	for T a type;
(b) $(T_1,...T_n)$	-> T	\
(c) $(T_1,...T_n)_p$	-> T	| for T, $T_1,...T_n$ types, $n \geq 0$.
(d) $(T_1,...T_n,...)_p$	-> T	/

In all of the cases, the type of the result (of the function call operator) is T. Its mode is **value**.

In cases (a) and (b) the Arguments may be of any type. Even though the types of the function's parameters are specified in case (b), a language processor may at most warn if the type of an argument does not match the type of a parameter.

In case (c), there must be n Arguments. The type of the i^{th} Argument must be assignment-compatible with T_i.

Case (d) is the same as case (c) except there may be more than n Arguments. The additional Arguments may be of any type.

Semantics

The T1 and the Arguments are evaluated in an unspecified order. The end of the evaluation of the Arguments is a sequence point. The end of the evaluation of the call is a sequence point.

For each of the Arguments A of type T_A and corresponding parameter type T, A is converted as if by assignment to a variable of type T. When T is unknown, which occurs in cases (a) and (b), and for the additional Arguments in case (d), T is implied by T_A as follows. T is

- Signed-Int if T_A is Signed-Char or Signed-Short-Int;
- Unsigned-Int if T_A is Unsigned-Char or Unsigned-Short-Int;
- Double if T_A is Float; or
- T_A otherwise.

Note that since an expression of type [?]:T' or [V]:T' is converted to type *T', T_A can never be of an array type [?]:T' or [V]:T'. Likewise, T_A can never be a functionality type T', since it is converted to *T'.

The function is called with its parameters taking on the converted values of the Arguments. The call is by value: copies of the values are passed. Thus, assignments into a function's parameters within the function body do not affect the Arguments passed.

Direct and indirect recursive calls to any function are permitted.

Discussion

E instead of EL is used in the rule for Arguments to avoid an ambiguity.

Early C compilers never made any check for parameter correspondence. Arguments of integral type were simply widened, and Floats were converted to Double. The specification of parameter types in a Function_definition *had no effect* on argument passing.

To preserve these semantics, a functionality type F affects argument passing only when F is a prototype functionality. When using a prototype, short integer and Float arguments can be passed more efficiently; the old semantics requires the possibly expensive conversion of Float to Double and the shortening of integers.

Furthermore, an integer argument may be passed to a function receiving a Float parameter, and the argument is conveniently converted to Float. Essentially, prototype functionalities are C's concession to the safer procedure call semantics of Pascal. *Example:*

Inefficient:
```
int f(x, c) float x; short s; {...}
f(3.2, 3);
```

Efficient:
```
int f     (float x, short s) {...}
f(3.2, 3);
```

In the first case, a Double and Signed-Int are passed and f's prologue converts the Double back to a Float and the Signed-Int back to Signed-Short-Int. In the second case, a Float and Signed-Short-Int are passed and no conversion occurs in f's prologue.

X3J11 provides that the scope of an implicitly-defined function F is the innermost block containing the call to F. We have instead made its scope the entire program. This prohibits an inconsistent declaration of F from appearing at the global lever later, such as in

```
main () {    /* Implicit declaration of f */
   f();      /* as extern int f();        */
   }
long f() { ...}
   /* Illegal:  long != int. */
```

Note that the declaration "**int** f();" and "**int** f(...);" are equivalent. The unfortunate duplication of syntax is due to compatibility with "old C".

8.25 Array Indexing: [ ] ---------- •

[pointer arithmetic]

Syntax

```
T1
-> T1 '[' EL  ']'
```

Constraints and Semantics

Same as for (* ((T1) + (EL))).

Discussion

[] is intended to be used to subscript an array. Because of the semantics of + in the equivalent expression (* ((T1) + (EL))), T1 can either denote an array or a pointer to an array; if the former, it is immediately converted to a pointer to the first element of the array. The +(EL) moves to the desired element of the array, and the outer * extracts the value. The reader may wish to verify that the mode of the result will always be **var**.

Therefore, if P is a pointer to the first element of array A, P[EL] and A[EL] denote the same object. Also, (EL)[A] and A[EL] are the same.

A[I, J, K] has unexpected semantics. It is the value of A[K] after I and J have been evaluated. Do not read A[I, J, K] as subscripting a three-dimensional array: there are only one-dimensional arrays in C.

8.26 Pointer Dereference and Member Selection: -> •

Syntax

```
T1
-> T1 '->' Member:Name
```

Constraints and Semantics

Same as for (*(T1)).Name.

Discussion

This notation is provided as a shorthand for the cumbersome notation (*p).f, where p is a pointer to a structure or union value and f is the name of a member of the value. The weak binding of the operator * makes necessary the () around the *p, for *p.f means *(p.f).

8.27 Member Selection: . ---------- •

[sizeof, &; **struct**, **union**]

Syntax

```
T1
-> Primary
-> T1 '.'  Member:Name;
```

Constraints

The type of T1 must be of the form Struct{M} or Union{M}. In particular, no incomplete structure or union types are permitted. The type must have a Member m named Name.

The type of the result is the type T of m, unless m is of an array type [...]:T and the selection is not an argument to **sizeof** or &, in which case the type of the result is *T.

If the latter case holds or T1 is of mode **value**, the result is of mode **value**. Otherwise, if the mode of m is **field**, the resulting mode is **field**; otherwise the resulting mode is **var**.

See Subsections *Pointer Reference* and ***sizeof*** for the reasoning behind not converting type T to *T in the presence of **sizeof** or &.

Semantics

The Primary is evaluated. The result is the value of the designated structure or union Member, except when T is an array type, in which case the result is the address of the first element of the array.

In general, a Member of a union may not be inspected unless the value of the union was assigned using that same Member, with the following exception: if a union contains several structures that share a common initial sequence, and if the value of the union was assigned using one of those structures, it is permitted to inspect the common initial part of any of them. (This follows X3J11.) ***Example:***

```
union  { struct {int Type;                  } N;
         struct {int Type; int IntNode;     } NI;
         struct {int Type; int FloatNode;   } NF;
       } U;
...
U.NF.Type = 1;      U.NF.FloatNode = 3.14;
...
if (U.N.Type == 1) U.NF.FloatNode = -U.NF.FloatNode;
```

8.28 Overriding Operator Precedence: ( ) ------- •

Syntax

```
Primary  -> '(' EL ')';
```

Constraints and Semantics

Same as for EL.

Discussion

Mere parenthesization does not change the constraints or semantics of an expression. For example, if EL is of mode M and type T, so is the result.

8.29 <IDENTIFIER>s

[**sizeof**, &; conversion of arrays to pointers; <IDENTIFIER>]

Syntax

```
Primary  -> Name: '<IDENTIFIER>';
```

Constraints

The Name must denote an object of some type T and mode M in the ordinary name space. If T is of the form [...]:T' and the Name is not an argument to **sizeof** or &, the type of the result is *T'; if it is of a functionality type T and mode **fcn**, the type of the result is *T; the mode of the result is **value**. Otherwise the type of the result is T and its mode is **var**.

See Subsections *Pointer Reference* and ***sizeof*** for the reasoning behind not converting type T to *T in the presence of **sizeof** or &.

Semantics

The value of the Primary is generally the object denoted by the Name. However, if T is an array type, the value is a pointer to the first element of the array; if a functionality type, the entry point of the function.

Discussion

Primary directly produces <IDENTIFIER> rather than the nonterminal Name so that <TYPEDEF_NAME>s are disallowed in expressions. The mode of an <IDENTIFIER> can never be **typedef** since any such an <IDENTIFIER> is instead in the lexical class <TYPEDEF_NAME>; therefore the constraints need not prohibit mode **typedef**.

8.30 Constants

[<INTEGER>, <FLOAT>, <CHAR>, <OCTAL>, <HEX>, <STRING>; string terminator; arrays of characters]

Syntax

```
Primary
-> Constant;
Constant
-> '<INTEGER>'|'<FLOAT>'|'<CHAR>'|'<OCTAL>'|'<HEX>'
-> '<STRING>'+;
```

Constraints

The mode of each Constant is **value**. Other constraints detailing each Constant's type can be found in Section *Lexicon.*

The constraints of two or more <STRING>s appearing in sequence are equivalent to a single <STRING> whose text is that of the individual <STRING>s concatenated, but without a separating '\000'.

Semantics

The value of two or more <STRING>s appearing in sequence is the same as that of a single <STRING> whose text is that of the individual <STRING>s concatenated, but without a separating '\000'. A single '\000' is appended to the concatenation and the result is the value of the <STRING>s, as if a single <STRING>.

Strings do not share storage with each other, even when written identically.

Discussion

Strings are arrays of characters, and as arrays are usually converted to a pointer to the literal. See Section *Lexicon.*

String concatenation is contained only in X3J11 and this definition.

8.31 Cast Types and Abstract Declarators -------- •

[Abstract_declarator]

Syntax

```
Cast_type
-> Type_specifiers Abstract_declarator?
;
```

Constraints

The occurrence of a Cast_type is associated with a type T' that is specified in the Type_specifiers preceding the optional Abstract_declarator. The type T of the Cast_type is T' if the Abstract_declarator is not written; otherwise, it is *Type* (T',A), where A is the Abstract_declarator and the definition of *Type* may be found in Section *Declarations/Declarators.*

Semantics

None. This is purely a type-specifying construct.

Discussion

In C the type of an object is specified both in Type and Declarator. For example, a pointer type is constructed in a Declarator, not a Type. But since a Declarator always declares a name, it is not possible to use it to describe a type without declaring a name.

Hence the invention of the Abstract_declarator. It is like a Declarator except the name is missing. The type of the missing name is the type denoted by the Abstract_declarator.

The unfortunate near-duplication of syntax in Abstract_ declarator could have been avoided were all type information provided in Type, rather than splitting it up into Type and Declarator. Then '(' Type ')' could have been used for a Cast_type.

8.32 Names

[<IDENTIFIER>, <TYPEDEF_NAME>]

Syntax

Name -> '<IDENTIFIER>' | '<TYPEDEF_NAME>';

The reason for '<TYPEDEF_NAME>' as an alternative for Name is that an identifier already declared of mode **typedef** is lexically a '<TYPEDEF_NAME>' instead of an '<IDENTIFIER>'. Thus, consider:

```
typedef int T;          /* Occurrence 1 of T. */
main() {
   T x;                 /* Occurrence 2 of T. */
   int T;               /* Occurrence 3 of T. */
   }
```

Occurrence 3 is a redeclaration of T as a variable of type Signed-Int. Due to the outer **typedef** declaration, occurrence 3 is a <TYPEDEF_NAME>. This essentially reflects a simple implementation of a C processor front-end: the lexical processor looks up each <IDENTIFIER> to see if it is a <TYPEDEF_NAME>, and if so, changes it to a <TYPEDEF_NAME>. The <TYPEDEF_NAME> lexical class is necessary for the proper parsing of declarations using <TYPEDEF_NAME>s, exemplified by occurrence 2.

Constraints and Semantics

None. The constraints for a Name are given where Name is used in the syntax.

8.33 Constant Expressions

[array size; bit-field length; enumeration literal; case constant; initialization]

In several places the C language requires an expression E that evaluates to a constant. Repeated here for convenience, they are: the size of an array or bit-field, the value of an enumeration constant, a case constant, and the initialization

of an object of storage class **static**. All but the last case are constrained to be of an integral type.

The following subset of well-formed expressions are constant expressions when each E is a constant expression:

- E ? E : E
- E **op** E, when **op** is one of the binary operators

  `*  /  %  +  -  << >>`

  `<  >  <=  >=  ==  !=`

  `&  ^  |  &&  ||`
- **op** E, when **op** is one of the unary operators

  `+  -  ~  !  sizeof`
- (Cast_type) E
- (E)
- <IDENTIFIER>, when declared as an enumeration literal
- any of <INTEGER> <CHAR> <OCTAL> <HEX> <FLOAT>

For constant expressions required to be of an integral type, <FLOAT> is excluded, and Cast_type must denote an integral type.

Constant Initializers can also employ

- the application of & to objects of storage class **static** and to arrays of storage class **static** subscripted with a constant expression
- an identifier declared of mode **fcn**
- an identifier declared of an array type

Appendix A
Language Extensions

A.1 Introduction

This section presents features in the High C language not typically found in implementations of C. These extensions fall into three categories:

- extensions that X3J11 has made to C,
- simple High C extensions, and
- radical High C extensions.

The first two classes of extensions are documented in the main body of this reference work. The last is documented only here — not, however, because the radical extensions are unimportant. Rather, they are too different to include in the main body of the language definition. Hence they were relegated to this appendix so that those readers not interested in them would not to have to skip over them.

For completeness, the first two classes of extensions are discussed briefly in Subsections A.2 and A.3. The third class is treated at length in Subsections A.4 through A.7. Finally, the last subsection is a brief tutorial on the X3J11 extension, function prototypes.

A.2 X3J11 Extensions to C

["\" as line continuator; string concatenation; constant suffixes; escape sequences; signed; long double; aggregate initialization; function prototypes]

- The use of '\' as a line continuator character — Section *Lexicon.*

  To overcome source line limitations, any line ending in the character '\' is treated as contiguous with the following line. Therefore any C word may be broken across line boun-

daries. This is most useful for multiple-line macro definitions and long strings, although there is a better solution for the latter — see the next item.

- String-constant concatenation — Section *Lexicon.*

  Juxtaposed string constants denote the concatenation of the constants. *Example:*

```
char *p = "Hi "  "there, "
          "folks.";
```

  is equivalent to

```
char *p = "Hi there, folks.";
```

  This feature is useful for long string constants that span more than a line of text, and for aligning portions of a string to emphasize or illuminate correspondences.

- Vertical tab as a delimiter — Section *Lexicon.*

  The ASCII vertical tab character is semantically equivalent to a blank when not appearing within a string or character.

- Suffixes u (U), l (L) in integer constants — Section *Lexicon.*

  These suffixes, either alone or together, in either order and independently in either upper or lower case, are permitted in integer constants: decimal, octal, and hexadecimal. The effect of "u" is to change the type T that the integer constant would otherwise have to the unsigned variety of T. The effect of "l" is to change T to the long variety of T. Thus, for example, "123" has type Signed-Int, "123u" Unsigned-Int, "1231" Signed-Long-Int, and "123ul" and "1231u" Unsigned-Long-Int.

- Suffixes f (F), l (L) in float constants — Section *Lexicon.*

  The suffix l, in upper or lower case, is permitted in floating-point constants. Its effect is to make the type of the constant Long-Double instead of the default Double. The suffix f, in upper or lower case, makes the constant's type Float rather than Double. f and l may not appear together.

- Escape sequences \a, \v, and \xddd in strings and characters — Section *Lexicon.*

  In addition to the escape sequences \n, \t, \b, \r, \f, \\, \', \", and \ddd (octal digits) allowed by KR, the sequences \a, \v, and \xddd are allowed, where \a denotes "audible alert" or the ASCII BEL character, \v denotes the ASCII vertical tab character, and ddd is a sequence of one to three hexadecimal digits (with each letter in upper or lower case): \xddd denotes a single byte whose value is ddd_{16}.

- The new reserved word **signed** — Sections *Lexicon* and *Declarations.*

  The type modifier **signed** may be used to indicate that the modified type is to be signed. This is most useful in guaranteeing that a **char** type is signed, since an implementation is free to decide whether the unadorned type denotation **char** denotes Signed-Char or Unsigned-Char. For example, "**signed char** c;" declares c of type Signed-Char, whereas "**char** c;" may be signed or unsigned depending upon the implementation.

- New type denotation **long double** — Section *Declarations.*

  The type specifier **long double** denotes a new floating-point type Long-Double having precision and range no less than type Double. This may be employed to obtain a triple-precision floating-point type.

- Initialization of automatic aggregates with static expressions — Section *Declaration/Non-Function Definitions.*

  An aggregate object O with storage class automatic may be initialized with the same initializing expressions permitted for static aggregates and additionally, if O is a structure or union, with a single expression of O's type.

- Initialization of structures with structure-valued expressions — Section *Declaration/Non-Function Definitions.*

A structure (**struct/enum**) may be initialized with a structure-valued expression, as in

```
extern struct s{int x;} f();
main() {
   struct s S = f();
   }
```

- Function prototypes — Sections *Declarations* and *Expressions.*

  See Subsection A.8 for an introduction to this Pascal-like concept in C.

A.3 High C Extensions Documented in the Manual Body

[underscores in numbers; intermixing statements and declarations; case ranges; aggregate initialization; pragma]

- Underscores in numbers — Section *Lexicon*

  Numbers — both floating-point and integer constants — may be written with the character '_' among the digits. Generally '_' takes the place of the English comma in numbers. *Example:*

  ```
  1_000_000 /* One million. */
  ```

- All characters in identifiers are significant — Section *Lexicon.*

- The ability to intermix declarations and statements — Section *Statements.*

  In a compound statement, i.e. the list of declarations and statements enclosed within { and }, declarations and statements may be interleaved, as opposed to declarations preceding statements. This allows one to place a declaration near its first use, where its initial value may be available. *Example:*

  ```
  int A[10];
  Compute_array(A);
  int M = Max(A, 10);
  ```

With this relaxed order, it is now possible to "execute" — formally, *elaborate*, a term borrowed from Ada — a declaration more than once. Elaboration includes initialization. *Example:*

```
Loop: ;
    int K, Count = 1;
    for (K = 1; K <= 10; K++) {
        ...Do_something();...
       }
    goto Loop;
```

Each time around the Loop, Count is initialized to one.

- Case ranges in the **case** Statement — Section *Expressions.*

  High C permits the extension "**case** E1..E2:" where the meaning is equivalent to "**case** E1: **case** E1+1: **case** E1+2: ... **case** E2:". *Example:*

```
switch (Ch) {
   case 'A'..'Z': Scan_id();          break;
   case '0'..'9': Scan_number();      break;
   default:       Scan_delimiter();   break;
   }    /* The latter break for safety: */
        /* in case of reordering cases. */
```

- Initialization of automatic aggregates with arbitrary expressions — Section *Declarations.*

  High C permits automatic aggregates to be initialized with arbitrary non-constant expressions, where X3J11 restricts initialization lists to constant expressions.

- New reserved word **pragma** — Sections *Lexicon, Declarations,* and *Statements.*

A.4 Named Parameter Association

[function call; positional parameters]

Functions declared with parameter names can be called with the named parameter association syntax of Ada. Such calls refer to the parameter names rather than their positions, so that the ordering of supplied parameters is irrelevant. The syntax is like that of a normal function call except that each actual parameter expression is preceded by the corresponding formal parameter name followed by "=>". *Example:*

```
typedef enum{Red, Green, Blue} Color;
void P(int A, float B, Color C, Color D) { ... }
...
P(C => Red, D => Blue, B => x*10.0, A => y);
```

One may also start the function call using positional parameter notation and switch to named association as the parameters are written down from left to right. Switching back to positional notation is not allowed, nor is any other variation. *Example:*

```
void Plot(Xlo, Xhi, Ylo, Yhi, Xinc, Yinc)
   float  Xlo, Xhi, Ylo, Yhi, Xinc, Yinc; {
   ...
   }
...
Plot(Alo, Ahi, Blo*2.0, Bhi*2.0, Yinc=>y, Xinc=>f(x+z));
```

The formal definition of this construct follows.

Syntax

To permit the => operator, add to the lexical grammar (Section *Lexicon*), the rule

```
Other_op -> '=' '>'                                    =>'<AS_IS>';
```

Add to the phrase-structure grammar (Section *Expressions/Function Call*), the rule

```
Arguments
-> Named_parameter_association:
   Unnamed_arguments:   (E ',')*
   Named_arguments:     ('<IDENTIFIER>' '=>' E) list ',';
```

Constraints

As in Section *Expressions/Function Call,* let F be the type of the function being called. F must be one of the two forms (see Section *Concepts*):

(b) $(T_1,...T_n)$ -> T \
(c) $(T_1,...T_n)_p$ -> T / for T, $T_1,...T_n$ types, $n \geq 0$.

Furthermore, in the declaration of the type F, all parameter names must be provided.

These requirements exclude functions declared as in "`int f();`" and "`void h(int i, float);`" from being called using named parameter association.

The <IDENTIFIER>s in Named_arguments must collectively be the names of distinct parameters whose values are not supplied positionally. Values for all parameters must be supplied through either the Unnamed_arguments (positionally) or the Named_arguments.

Given these constraints, it is possible to transform the function call into a purely positional form. Further Constraints and Semantics are then as if the Arguments were thus transformed.

Semantics

Since the Constraints detail how named parameter association can be transformed into positional form, the Semantics here are the same as the Semantics of the transformed call discussed in Section *Expressions.*

A.5 Nested Functions and Full-Function Variables

[Pascal; function parent and environment; up-level addressing; static link; display; function address versus full-function value; "!"]

In High C, functions may be defined within functions. Such functions are called *nested.* This facility endows High C with an expressive power that is found in Pascal.

In the body of a nested function N, any name in a containing scope may be used. That is, the body of N may use names local to other functions that contain N. This is called *up-level referencing* and any such names are said to be *up-level referenced* from N. The single restriction is that **register**-class variables may not be up-level referenced.

Up-level referencing may be achieved by making available to N, at each call to it, a way to reference the collection of locals of each of its enclosing functions. This reference method is called N's *environment.* The function P immediately enclosing N is called N's *parent;* the next enclosing function G its *grandparent;* and so on. The collection of locals of each function is called its *stack frame.*

(**Technical note:** N's environment may be implemented by passing to N, at each call to it, a "hidden" parameter that is a reference to P's stack frame. If this is done for all functions, P will have in its stack frame a similar hidden parameter that links it to G, and so on out to the global level where functions need no such link. This is called the *static link* method of implementing up-level referencing, the method of choice for best efficiency and optimization possibilities, as opposed to the *display* method, which is not described here.)

Therefore, a major difference between nested functions and non-nested functions is that the address of a nested function N does not entirely capture N's "value": the environment is also required. In contrast, the address of a non-nested or global function G entirely captures its value, since G has no parent and thus needs no environment. The C notion of "pointer to function" is therefore sufficient to capture the value of G but not that of N.

Hence High C disallows taking the address of a nested function, and, where C assumes & before any expression of type function, High C does *not* assume the & if the expression is of a nested-function type.

Full-function values. We refer to the combination of a function address and its environment as a *full-function value*, as opposed to just a "function address".

All of the capabilities associated with global functions, such as passing them as parameters and storing their value into variables, is available for nested functions, although new syntax is required.

A variable capable of holding a full-function value, and therefore the value of a nested function, is declared as a function declaration would be, except that "!" follows the parenthesized formal parameter list. *Example:*

```
int ffv()!;
```

In contrast, a standard C variable capable of holding only a function address is declared using the pointer syntax:

```
int (*fa)();
```

`ffv` may be called with the expression "`ffv();`", but *not* with "`(*ffv)()`", since `ffv` is *not* (just) a pointer.

Any nested function may be assigned to `ffv`. A global function `G` may be assigned to `ffv` *by dereferencing it*, since of course `G` is transformed to `&G` by the compiler and must be dereferenced to obtain the full-function value of `G`, not just its address: "`ffv = *G;`". The environment stored in `ffv` in such an assignment is meaningless, since `G` needs no environment. Upon calling the value in `ffv`, the environment is passed to `G`, but `G` (indeed every global function) safely ignores it.

Nested functions may be passed as parameters: the full value is passed. The full-function value of a global function may be passed by dereferencing the global function: the passed environment is meaningless.

An argument can be declared as being a full-function value by using the new syntax:

```
int f(ffv) int ffv()!; { ... ffv(); ...}
```

Only the names of function constants may be dereferenced to produce full-function values. The dereference of a pointer to a function is immediately converted back to an address by standard C rules for function expression conversion. Thus:

```
extern int sub();
main () {
   int (*fa)();
   int Nested() {...}
   main(*main);      /* Passes the full value of main.      */
   main(*sub);       /* Passes the full value of sub.       */
   main(*fa);        /* Passes fa,  since *fa => &*fa = fa. */
   main(Nested);     /* Passes the full value of Nested.    */
   }
```

This extension is compatible with ANSI standard C since the dereference of an expression of type pointer-to-function is permitted only in the context of an expression denoting a function to be called, e.g. "(*fa)()"; but "(*fa)(*fa)" is illegal in ANSI C.

Example: As an example of the use of full-function values, we present a call to a sort function that takes as parameters two functions:

```
extern  void Quick_sort(
   int  Lo, int Hi, int Compare(int a,int b),
   void Swap(int a,int b)
   );
static Sort_private_table() {
   Entry Entries[100];
   int Compare(int a,int b) {
       return Entries[a] < Entries[b];
       }
   void Swap(int a,int b) {
      Entry Temp = Entries[a];
      Entries[a] = Entries[b]; Entries[b] = Temp;
      }
```

```
...
Quick_sort(1, 100, Compare, Swap);
}
```

Here it is necessary for Compare and Swap to be local to Sort_private_table since the table Entries is local to that function. In standard C, Entries, Compare, and Swap would have be moved outside of Sort_private_table. This works fine in this simple case, but if Sort_private_table were recursive, one would have to explicitly manage a stack of Entries arrays to get the desired effect.

Although this example may seem contrived, a stripped-down version of a practical Pascal program, translated into High C, is included in distributions of High C compilers. The name of the file is "analyze.c" and it implements a graph traversal algorithm. Any C programmer that thinks standard-C function capabilities are adequate should read this program and attempt to translate it to standard C.

Casting any full-function value of one type to any full-function value of another type is permitted, in concert with the ability to cast function addresses in standard C. The **sizeof** a full-function type may be taken and is always greater than the **sizeof** a function address, since the former includes the environment.

The additional syntax required to permit the declaration of full-function types follows, and is simply a repeat of the rules for standard function syntax except that "!" is allowed:

```
Declarator'
-> Extended_function_specification_declarator:
   Declarator' Parameters '!';
Abstract_declarator'
-> Abstract_declarator'? Abstract_parameters '!';
Declarator2'
-> Declarator2' Parameters '!';
```

A.6 Communication with Other Languages

[pragmas Calling_convention, Data, Code]

A High C module can communicate with modules written in other languages partially by virtue of the pragmas Calling_convention, Data, and Code. Although the syntax of these pragmas is machine independent, their effects are sometimes machine dependent, and hence their documentation can be found in the **Programmer's Guide**, Section *Externals.*

A.7 Intrinsics

[_abs, _min, _max; _find_char, _skip_char, _fill_char; _move, _move_right, _compare]

High C contains a set of so-called "intrinsic functions" that supply: (a) the ability to take the absolute value, minimum, and maximum of values of any arithmetic type, and (b) the ability to move and compare bytes of memory using the host machine's most efficient instructions. Intrinsic functions need not be declared to be used. Below is a list of the intrinsic functions followed by their descriptions in the same order as the list.

```
_abs          _max          _min

_find_char    _skip_char
_move         _move_right
_fill_char    _compare
```

_abs(x)

x must be of an arithmetic type. The result is the absolute value of x, of the same type as the type of x.

_max(e1, e2, . . .)
_min(e1, e2, . . .)

e1,e2,... must be of arithmetic types. The result is the maximum/minimum of e1,e2,... .

More precisely, _max(e1, e2) is of type T = Common(type of e1, type of e2) and is the maximum value of e1 and e2, each casted to type T. _max(e1, e2, ..., en), where n ≥ 3, is (recursively) defined to be _max(_max(e1, e2), e3, ..., en). The specification is similar for _min. *Example:*

```
float f; unsigned long ul; int i;
main () {
    _min(f, ul, i);   /* Has type Float. */
    _min(ul, i);      /* Has type Unsigned-Long-Int. */
    _min(i, f);       /* Has type Float. */
    }
```

It is guaranteed that the operands of _max and _min are evaluated at most once, unlike the standard macro definition of max (min), e.g. "**#define** max(x, y) ((x)>(y)?(x):(y))", where one argument is evaluated once and the other twice.

```
unsigned
_find_char(any_pointer p,
           unsigned search_length, char search_char)
```

Searches ((char*)p)[0] through ((char*)p)[search_length -1] for search_char using the most efficient host instruction to do so. It returns the index i (a number in the range 0.. search_length-1) if it found the character, where p[i] = search_char; otherwise it returns search_length, indicating a failed search.

The standard function strlen can be implemented using _find_char:

```
#define strlen(s) _find_char(s, 65_535, 0)
```

assuming at most 65,535 characters in a string. This version of strlen generates in-line code.

```
unsigned
_skip_char(any_pointer p,
           unsigned search_length, char search_char)
```

Does the same as _find_char, except that it searches for the first character *not* equal to search_char.

```
void
_fill_char(any_pointer p, unsigned len, char fill)
```

Fills memory from ((char*)p)[0] to ((char*)p)[len-1] with the fill character. This can be used to implement the standard library function memset:

```
#define memset(p, fill, len) _fill_char(p, len, fill)
```

```
void
_move(any_pointer from, any_pointer to, unsigned len)
```

Moves len bytes from the address ((char*)from) to the address ((char*)to). The move occurs from left-to-right (lower-to-higher addresses). If from < to < from+len, use _move_right below.

This can be used to implement the standard library function memcpy:

```
#define memcpy(dest, src, len) _move(src, dest, len)
```

```
void
_move_right
     (any_pointer from, any_pointer to, unsigned len)
```

Does the same as _move except the move occurs from right-to-left (higher-to-lower addresses). If to < from < to+len, use _move above.

int
_compare(any_pointer p1, any_pointer p2, unsigned **len)**

Compares the string ((char*)p1)[0]..((char*) p1)[len-1] with the string ((char*)p2)[0]..((char*)p2)[len-1]. It returns 0 if they are identical, -1 if the first byte at which the two disagree is less in the first string than the corresponding byte in the other stream, and +1 otherwise.

A.8 Brief Tutorial on Prototypes

[argument widening and shortening; function call reliability]

X3J11 provides an alternate form for specifying the parameter types in function declarations. When functions declared in this form are called, the types of the arguments must be assignment compatible with the types of the formal parameters, and as in assignment, any necessary conversion to the formal parameter type is applied.

In contrast, KR specifies that the types of the declared parameters (if available) are irrelevant at the function call, and instead that default conversions are required: **char**s are widened to **int**s and **float**s to **double**s. But this can lead to incorrect and unpredictable results when the type of the value passed is not compatible with the declared type; e.g.:

```
float Min(f1,f2) float f1,f2; {
   return f1 < f2 ? f1 : f2;
   }
int i; float f,fmin;
fmin = Min(i,f);
```

Here an **int** i is passed to Min instead of a **double**. In addition if **sizeof(int)** != **sizeof(double)**, the f2 parameter to Min will not be passed in the correct location on the stack. (Recall that **double**s are always passed as parameters in KR, never **float**s.)

In "prototype" syntax the specification of a parameter name and its type are not separated and appear more like a

standard C declaration. The parameter declarations are separated by commas, just as in standard parameter lists:

```
float Min(float f1, float f2) {
   return f1 < f2 ? f1 : f2;
   }
int i; float f, fmin;
fmin = Min(i, f);
```

With the prototype syntax, two things happen: First, **float** values are passed, not **doubles**. This avoids the expensive widening-at-call and shortening-at-function-entry cost of KR. Second, the **int** i is converted to **float** before it is passed, just as if i were being assigned to a **float** variable.

Notice the repetition of **float** in the parameter list: no factoring of declarations is permitted, as in "**float** f1, f2;" due to syntactic constraints.

A prototype declaration may be used anywhere a standard declaration is allowed. In addition the parameter names may be omitted in declarations that are not definitions; e.g.:

```
float Min(float, float);
```

Prototype functionality is new, designed to allow some type-checking across separate compilation units. For more information, see Sections *Expressions/Function Call, Declarations/Function Definitions*, and *Concepts*.

Appendix B
Collected Grammar Rules

The grammar rules used in the various sections are collected here for easy reference.

Phrase-Structure Grammar

```
parser C_phrase_structure, C_conditional_compilation_expression:

C_phrase_structure          # The above is for the preprocessor.
-> External_declaration*
;
#####################################################################
# Declarations.                                                     #
#####################################################################

External_declaration
-> Unspecified_declaration:
   ( Function_definition | Non_function_definitions ';' )
-> Specified_declaration                    # With specifiers.
-> Pragma_call
-> ';'                                      # Syntactic oddity of KR.
;
Pragma_call
-> 'pragma' Name  ('(' (E list ',')? ')')?
;
Specified_declaration
-> Specifiers ';'
-> Specifiers Function_definition
-> Specifiers Non_function_definitions ';'
;
#####################################################################
# Types and Specifiers.                                             #
#####################################################################

Specifiers
-> Type_or_storage_classes
;
Type_or_storage_classes
-> Storage_class Type_or_storage_classes?
-> Typedef_reference: '<TYPEDEF_NAME>' Storage_class?
-> Type ASCs
-> Adjective ASCs (Type ASCs)?
;
ASCs
-> (Adjective | Storage_class)*
;
```

```
Storage_class
-> 'auto' | 'extern' | 'register' | 'typedef' | 'static'
;

Adjective
-> 'short' | 'unsigned' | 'long' | 'signed'
;

Type_specifiers
-> Typedef_reference:'<TYPEDEF_NAME>'
-> Adjective* Type Adjective*
-> Adjective+
;

Type
-> 'char' | 'int' | 'float' | 'double' | 'void'
-> Tagged_type
;

Tagged_type
-> Complete_definition:              'struct' Tag?  '{' Member_list  '}'
-> Complete_definition:              'union'  Tag?  '{' Member_list  '}'
-> Complete_definition:              'enum'   Tag?  '{' Literal_list '}'
-> Use_or_incomplete_definition: ( 'struct' Tag
                                 | 'union'  Tag
                                 )
-> Reference:                       'enum'   Tag
;

Tag
-> Tag:Name
;

Literal_list
-> (Name ('=' Constant:E)?) list ',' ','?
;

Member_list
-> Also_is_a_list:Members list ';' ';'?
;

Members
-> Type_specifiers (Structure_member list ',')?
;

Structure_member
-> Declarator
-> Field_member:Declarator? ':' Bits:Constant:E
;

###############################################################
# Declarators.                                                #
###############################################################

Declarator
-> '*' Declarator
-> Declarator'
;

Declarator'
-> Declarator' '[' Array_specification ']'
-> '(' Declarator ')'
-> Function_specification:
   Declarator' '(' Parameters ')'
-> Declared:Name
;
```

```
Array_specification
-> Constant:E?
;
Parameters
-> Parameter_names_only: Parameter_name list ',' More_parms?
-> Abstract_parameters
;
Abstract_parameters
-> (SD list ',' More_parms?)?
;
SD
-> Specifiers (Abstract_declarator | Declarator2)?
;
More_parms
-> ',' '...'?
;
Parameter_name
-> '<IDENTIFIER>'
;
Abstract_declarator
-> '*' Abstract_declarator?
->     Abstract_declarator'
;
Abstract_declarator'
-> Abstract_declarator'? '[' Array_specification ']'
-> Function_specification:
   Abstract_declarator'? '(' Abstract_parameters ')'
-> '(' Abstract_declarator ')'
;
# Declarator2(') needed to avoid an ambiguity.
Declarator2
-> '*' Declarator2
-> Declarator2'
;
Declarator2'
-> Declarator2' '[' Array_specification ']'
-> Function_specification:
   Declarator2' '(' Parameters ')'
-> '(' Declarator2 ')'
-> Declared_name:'<IDENTIFIER>'
;

#####################################################################
# Definitions.                                                      #
#####################################################################

Non_function_definitions
-> (Declarator ('=' Initializer)?) list ','
;
Initializer
-> E
-> '{' Initializer list ',' ','? '}'
;
Function_definition
-> Fcn:Declarator Parameter_types Compound_statement
;
```

```
Parameter_types
-> (Specifiers (Parameter:Declarator list ',')? ';')*
;

#################################################################
# Statements.                                                   #
#################################################################

Compound_statement
->  '{' (Specified_declaration | Pragma_call | Statement)* '}'
;
Statement
-> Compound_statement
-> EL ';'
                                        # Switch and its cases:
-> 'switch' '(' EL ')' Switch_body:Statement
-> 'case'   Case_label: # New from X3J11:
            Constant:E ('..' Constant:E)?
                       ':'  Statement
-> 'default'           ':'  Statement
                                        # End of switch and its cases.
-> 'if'     '(' EL ')'   Statement ('else' Statement)?
-> 'while'  '(' EL ')'   Statement
-> 'do'     Statement    'while' '(' EL ')' ';'
-> 'for'    '(' First:EL? ';' Next:EL? ';' Last:EL? ')'
                    Body:Statement
-> 'break'          ';'
-> 'continue'       ';'
-> 'return' EL?     ';'
-> 'goto'   Target_label:Name ';'
-> Labelled_statement:
   Label:Name       ':'  Statement
-> ';'
;

#################################################################
# Expressions.                                                  #
#################################################################

C_conditional_compilation_expression-> E2; # For the preprocessor.

OEL -> EL?;                             # Optional Expression List.
EL  -> E    -> E ',' EL;                # Equivalent to:  E list ','.
E   -> E1;                              # Two columns for compactness.
E1  -> E2   -> Lvalue:Term' Plain_assignment:'=' E1
            -> Lvalue:Term' (Assignment_operator E1);
            Assignment_operator
            -> '|=' | '^=' | '&=' | '>>=' | '<<='
             | '+=' | '-=' | '*=' | '/='  | '%=';
E2  -> E3   -> Conditional_expression:
               E3 '?'  EL ':' E2;
E3  -> E4   -> E3 '||' E4;
E4  -> E5   -> E4 '&&' E5;
E5  -> E6   -> E5  '|' E6;
E6  -> E7   -> E6  '^' E7;
E7  -> E8   -> E7  '&' E8;
E8  -> E9   -> E8 '==' E9
            -> E8 '!=' E9;
```

```
E9  -> E10  -> E9 '<'  E10
            -> E9 '>'  E10
            -> E9 '<=' E10
            -> E9 '>=' E10;
E10 -> E11  -> E10 '>>'E11
            -> E10 '<<'E11;
E11 -> E12  -> E11 '+' E12
            -> E11 '-' E12;
E12 -> E13  -> E12 '*' E13
            -> E12 '/' E13
            -> E12 '%' E13;
E13 -> Term
;
Term
-> Term'
-> '(' Cast_type ')' Term
;
Term'
-> '*' Pointer:Term
-> '&' Term
-> '-' Term
-> '+' Term
-> '~' Term
-> '!' Term
-> '^' Term
-> 'sizeof' ('(' Cast_type ')' | Term')
-> '++' Lvalue:Term
-> '--' Lvalue:Term
-> T1
;
T1
-> Lvalue:T1 '++'
-> Lvalue:T1 '--'
-> T1 '(' Arguments ')'
;
Arguments
-> (E list ',')?
;
T1
-> T1 '['  EL  ']'
-> T1 '->' Member:Name
-> Primary
-> T1 '.'  Member:Name
;
Primary
-> '(' EL ')'
-> Name:'<IDENTIFIER>'
-> Constant
;
Constant
-> '<INTEGER>' | '<FLOAT>' | '<CHAR>' | '<OCTAL>' | '<HEX>';
-> '<STRING>'+ Cast_type
;
```

```
Name
-> '<IDENTIFIER>' | '<TYPEDEF_NAME>'
;
end     C_phrase_structure
```

Preprocessor Phrase-Structure Grammar

```
parser C_preprocessor_text:

C_preprocessor_text
-> (Control_text | Word)*
;

Word  -> Any - '<CONTROL>' - '<C-EOL>'  # Any word from the lexical
;                                       # analyzer except these two.

Words -> Word*
;

Control_text
-> '<CONTROL>'!'include'  '<<STRING>>' '<C-EOL>'
-> '<CONTROL>'!'include'   '<STRING>'  '<C-EOL>'
-> '<CONTROL>'!'c_include' '<STRING>'  '<C-EOL>'
-> '<CONTROL>'!'define'
   ( Macro_name:'<NO_PARMS>'
   | Macro_name:'<WITH_PARMS>'
     '(' ( Parameter:'<IDENTIFIER>' list ',' )? ')'
   ) Body:Words '<C-EOL>'
-> '<CONTROL>'!'undef' Macro_name:'<IDENTIFIER>' '<C-EOL>'
-> ( '<CONTROL>'!'if'    E '<C-EOL>'   If:Words
   | '<CONTROL>'!'ifdef'  '<IDENTIFIER>' '<C-EOL>' Words
   | '<CONTROL>'!'ifndef' '<IDENTIFIER>' '<C-EOL>' Words
   )
    ('<CONTROL>'!'elif'  E '<C-EOL>' Elif:Words)*
    ('<CONTROL>'!'else'    '<C-EOL>' Else:Words)?
     '<CONTROL>'!'endif'   '<C-EOL>'
;

E     -> C_conditional_compilation_expression;  #  See the High C PSG.

#     The E nonterminal generates the same language as E2 in the C phrase-
# structure grammar (PSG: see Section Expressions and below), except that
# Primary is extended with the following two additional alternatives:

#     -> 'defined' Macro_name:'<IDENTIFIER>'
#     -> 'defined' '(' Macro_name:'<IDENTIFIER>' ')'

end     C_preprocessor_text
```

Lexical Grammar

```
scanner C_lexicon:

C_lexicon -> Text;
  Text    -> (Words Line_end)* (Control_line Text)?
          -> \Scanning Skipped_lines Control_line Text;

  Words   -> Word*;
  Word    -> String | Char | Number | Identifier
           | Delimiter | Punctuator | Operator | Comment;

Identifier-> Id_text                                          =>'<IDENTIFIER>';
  Id_text -> Letter (Letter | Digit)*;
  Letter  -> 'A'..'Z' | 'a'..'z' | '_' ;

Number    -> Integer | Octal | Float | Hex ;
Integer   -> '1'..'9' ('_'? Digits)? Integral_suffix?         =>'<INTEGER>';
Octal     -> '0' ('_'? Digits)? Integral_suffix?              =>'<OCTAL>';
Hex       -> '0' ('X' | 'x') Higits Integral_suffix?          =>'<HEX>';
Float     -> Mantissa Exponent? Float_suffix?                 =>'<FLOAT>'
          -> Digits   Exponent  Float_suffix?                 =>'<FLOAT>';
  Mantissa-> '.' Digits | Digits \Dot_dot '.' Digits?;
  scanner Dot_dot: Dot_dot -> '.' '.'; end Dot_dot
  Exponent-> ('E'|'e') ('+'|'-')? Digits;

  Integral_suffix -> 'u' 'l'? | 'l' 'u'? | 'U' 'L'? | 'L' 'U'?;
  Float_suffix    -> 'L'      | 'l'      | 'F'      | 'f';

  Higits  -> Higit+ list '_';                          # _ is non-standard.
  Higit   -> '0'..'9' | 'A'..'F' | 'a'..'f' ;
  Digits  -> Digit+ list '_';                          # _ is non-standard.
  Digit   -> '0'..'9' ;
  Oigits  -> Oigit+ list '_';                          # _ is non-standard.
  Oigit   -> '0'..'7' ;

String    -> '"'  DQchar* '"'                                 =>'<STRING>';
Char      -> '''  SQchar '''                                  =>'<CHAR>';
  DQchar  -> Any-'\'-'"'  | '\' Special ;
  SQchar  -> Any-'\'-'''  | '\' Special ;
  Special -> 'a' | 'b' | 'f' | 'n' | 'r' | 't' | 'v'
           | '\' | '"' | '''
          -> Oigit (Oigit Oigit?)?                            # Octal.
          -> 'x' Higit (Higit Higit?)?;                       # Hexadecimal.

Operator  -> AssignOp | OtherOp;
  OtherOp -> '-' | '&' '&' | '|' '|' | '>' '='
           | '<' | '=' '=' | '!' '=' | '<' '='
           | '>' | '+' '+' | '-' '-' | '-' '>'
           | '!' | '*'     | '.'     | '?'          =>'<AS_IS>';
  # Operators that can be followed by '=' in assignments.
  AssignOp->('^' | '>' '>' | '<' '<'
            |'+' | '*'     | '&'     | '%'
            |'-' | '/'     | '|'
            ) '='?                                            =>'<AS_IS>';
```

```
Punctuator-> '(' | ')' | '[' | ']' | '{' | '}' | ':' | ';' | ',' | '.' | '?' | '~'
                                                                    =>'<AS_IS>';

Deliniter -> ( Space:    ' '+  | HorizTab:'ht'
             | FormFeed:'ff'   | Vert:Tab:'vt'
             )+                                                     =>'<DELETE>';

Comment   -> '/' '*' Rest                                           =>'<DELETE>';
  Rest    -> Most* '*'+ ('/' | (Most-'/') Rest);
    Most  -> Any-'*' | Eol;


# Preprocessor lexicon:

Line_end        -> Eol                                              =>'<DELETE>';

Control_line    -> Sharp Deliniter? Control? Line_end;
  Sharp         -> '#'                                              =>'<DELETE>';
  Control       -> ( /Define_word Define Deliniter Macro
                   | (Include_text else Other_control)
                   ) Words Done;
  Done          ->                                                  =>'<C-EOL>' ;
  Other_control -> Id_text                                          =>'<CONTROL>';
  Macro         -> Id_text  \LP                                     =>'<NO_PARMS>'
                -> Id_text  /LP                                     =>'<WITH_PARMS>';
                scanner LP: LP -> '(' ; end LP
  scanner Define_word:
    Define_word -> 'd' 'e' 'f' 'i' 'n' 'e';
    end   Define_word
  Define        -> Define_word                                      =>'<CONTROL>';

  scanner Include_text:
    Include_text-> Include_word Deliniter? Funny_string;
    Include_word-> 'i' 'n' 'c' 'l' 'u' 'd' 'e'                      =>'<CONTROL>';
    Funny_string-> L_angle File_name R_angle;
    L_angle     -> '<'                                              =>'<DELETE>';
    R_angle     -> '>'                                              =>'<DELETE>';
    File_name   -> (Any-'\'-'"'-'>' | '\' Special)*                 =>'<<STRING>>';
    end   Include_text

Skipped_lines   -> (\Sharp Skipped_line)*;
  scanner Sharp:    Sharp -> '#'; end Sharp
  Skipped_line  -> Skip_suffix? Line_end;
  Skip_suffix   -> ( Not_special | Slash | Comment | DString )*;
  Not_special   -> (Any - '/'-'"')+                                 =>'<DELETE>';
  Slash         -> '/'                                              =>'<DELETE>';
  DString       -> String_text                                      =>'<DELETE>';

end     C_lexicon

reserve '<IDENTIFIER>' # Would-be ids used as key words in the PSG.

predefined Scanning     # External feedback from the preprocessor.
         # Eol          # Predefined in grammar notation:  end-of-line.
```

Appendix C
High C ™
Phrase-Structure Chart

The chart below was produced by the MetaWare Translator Writing System (TWS) from the grammar that also produces tables by which the High C compiler parses its programs. A few of the diagrams in the chart have been touched up by hand for increased efficiency of space usage. The interested reader should see the **MetaWare TWS User's Manual.**

To "read" the syntax diagrams of which the chart is composed just follow the "railroad tracks" to find out what words may be written in what order. Start at the upper left corner and go only with the arrows. Eventually you will be able to escape the tracks to the right, in which case a syntactically correct phrase will have been formed.

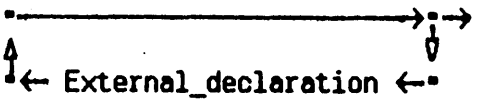
To form an entire program, start with the first phrase name diagrammed, namely High_C_phrase_structure. When another such name is encountered you must follow the tracks in its diagram and then return to the current tracks just after the phrase name that caused your departure. When a basic symbol is encountered, just write down the symbol.

The result of your tracks visitations and the writing down of the basic symbols will be a grammatically correct High C program. To find out how to form words correctly from individual characters see the "lexical chart" in the next appendix.

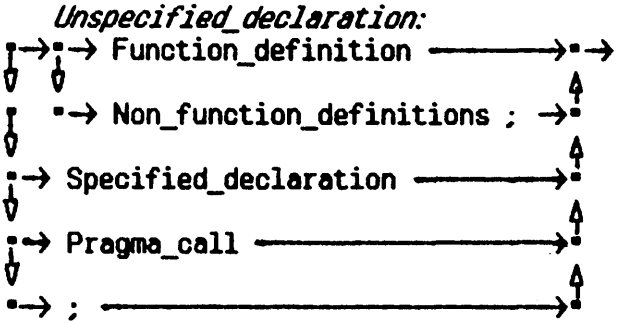
Here symbols like <IDENTIFIER> refer to the symbols described by the lexical chart; their names are usually self-explanatory, but when in doubt consult the following appendix.

The *italicized* words in these charts are purely commentary. Their removal would not change the language described by the chart. For example, *Constant* E (E for Expression) is equivalent to just E, but the intent is to inform the reader that in the current context the Expression must be constant.

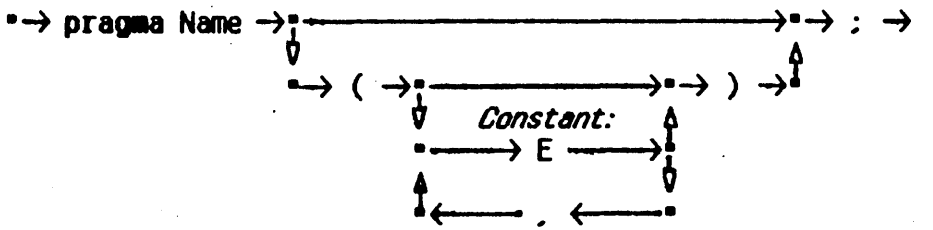
High_C_phrase_structure =



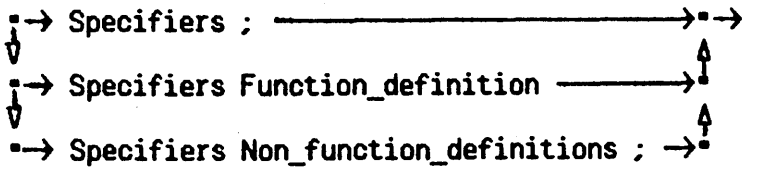
External_declaration =



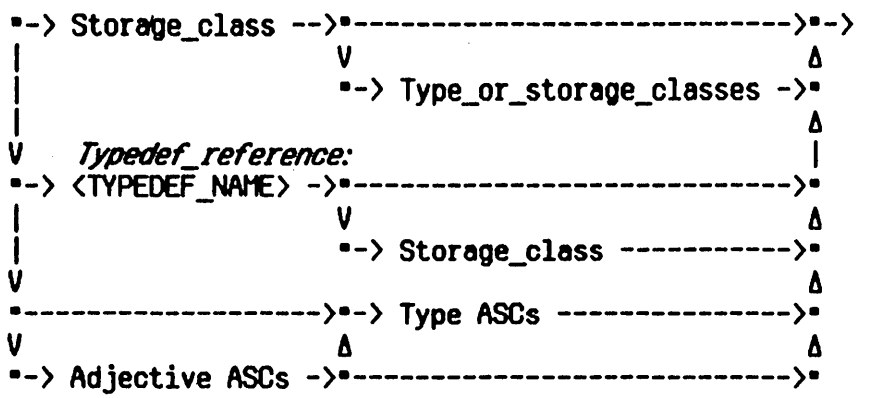
Pragma_call = /* Direct the compiler. */



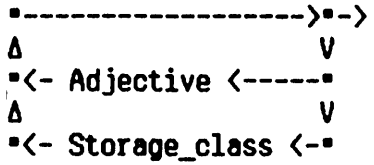
Specified_declaration =



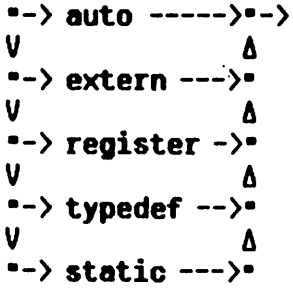
Specifiers =
Type_or_storage_classes =



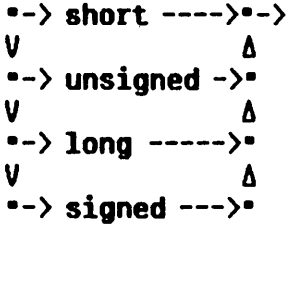
ASCs =



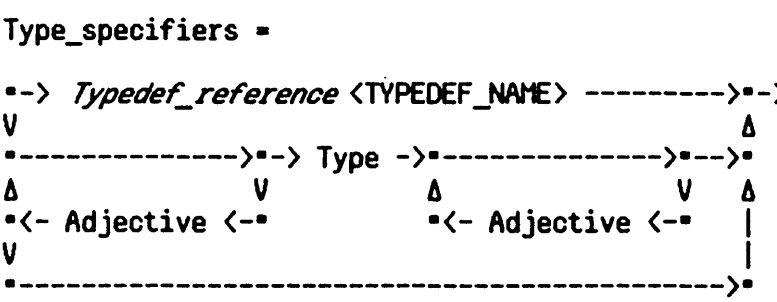
Storage_class =



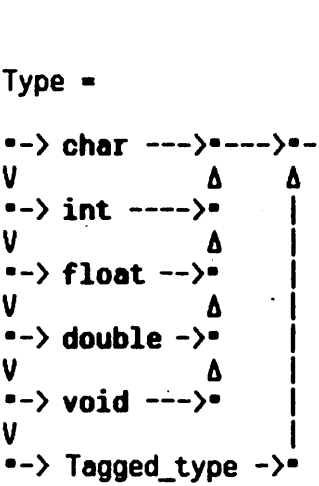
Adjective =



Type_specifiers =

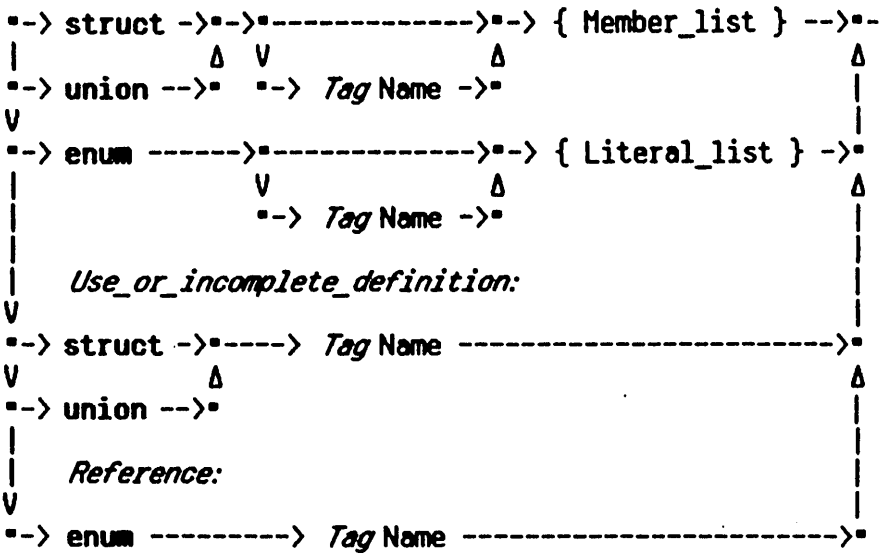


Type =

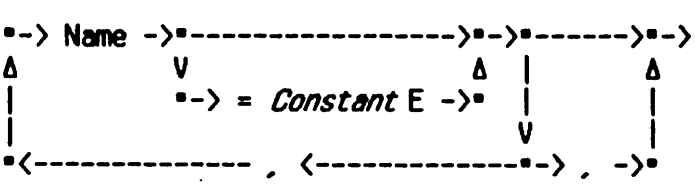


Tagged_type =

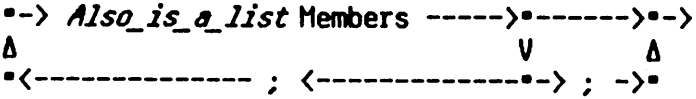
Complete_definition:



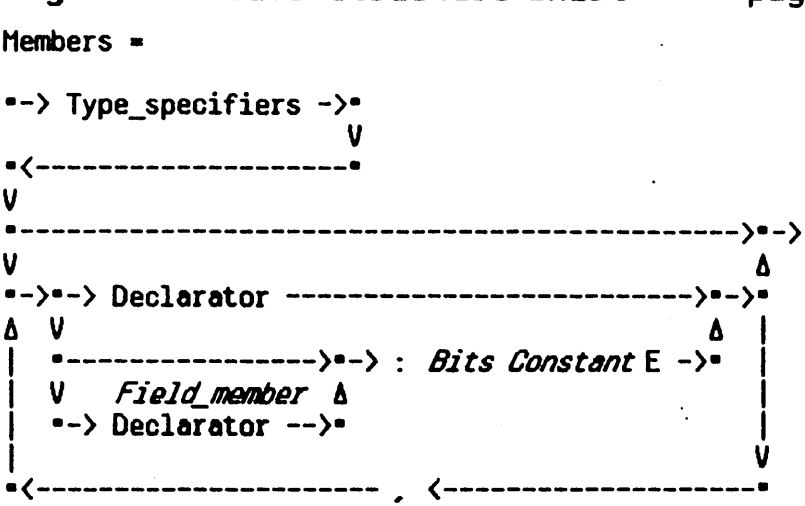
Literal_list =



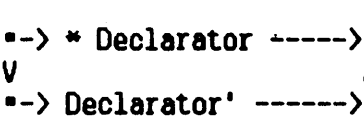
Member_list =



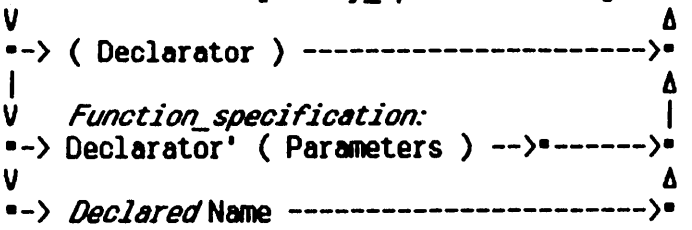
Members =



Declarator =

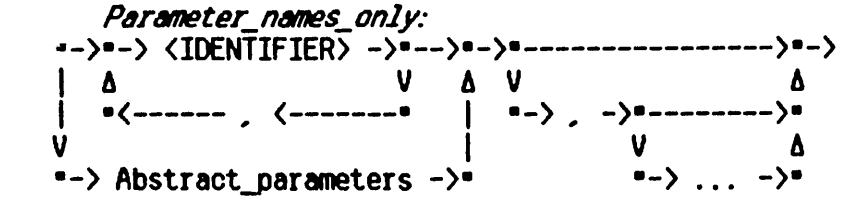


Declarator' =

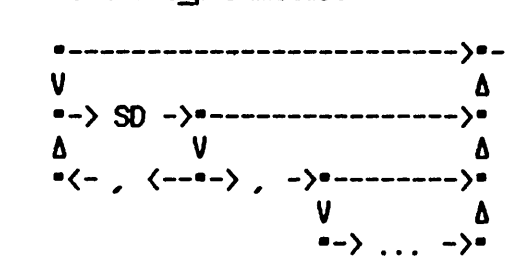


Array_specification =

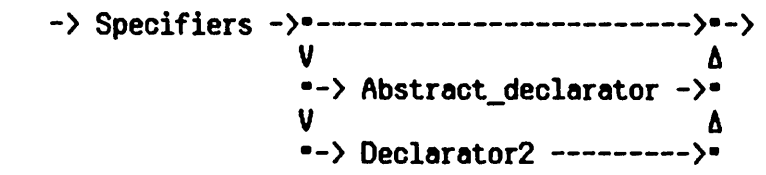
Parameters =



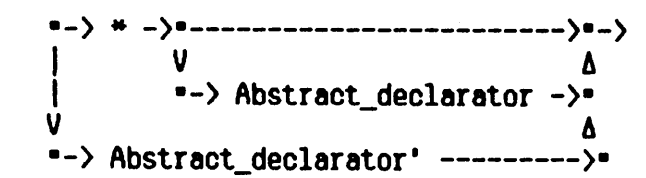
Abstract_parameters =



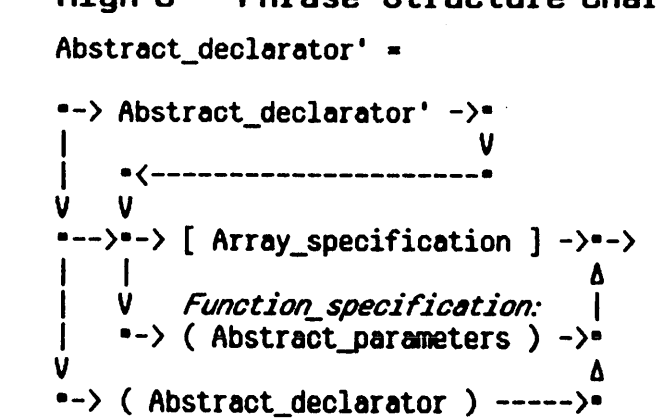
SD



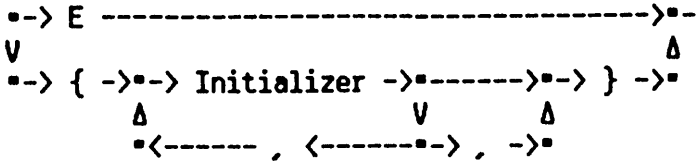
Abstract_declarator =



Abstract_declarator' =



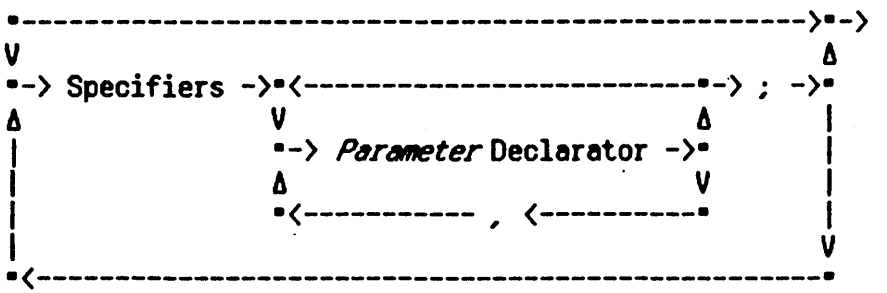
Initializer =



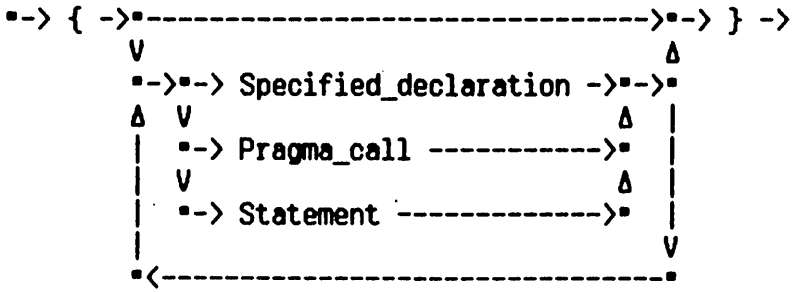
Function_definition =

*-> *Fcn* Declarator Parameter_types Compound_statement ->

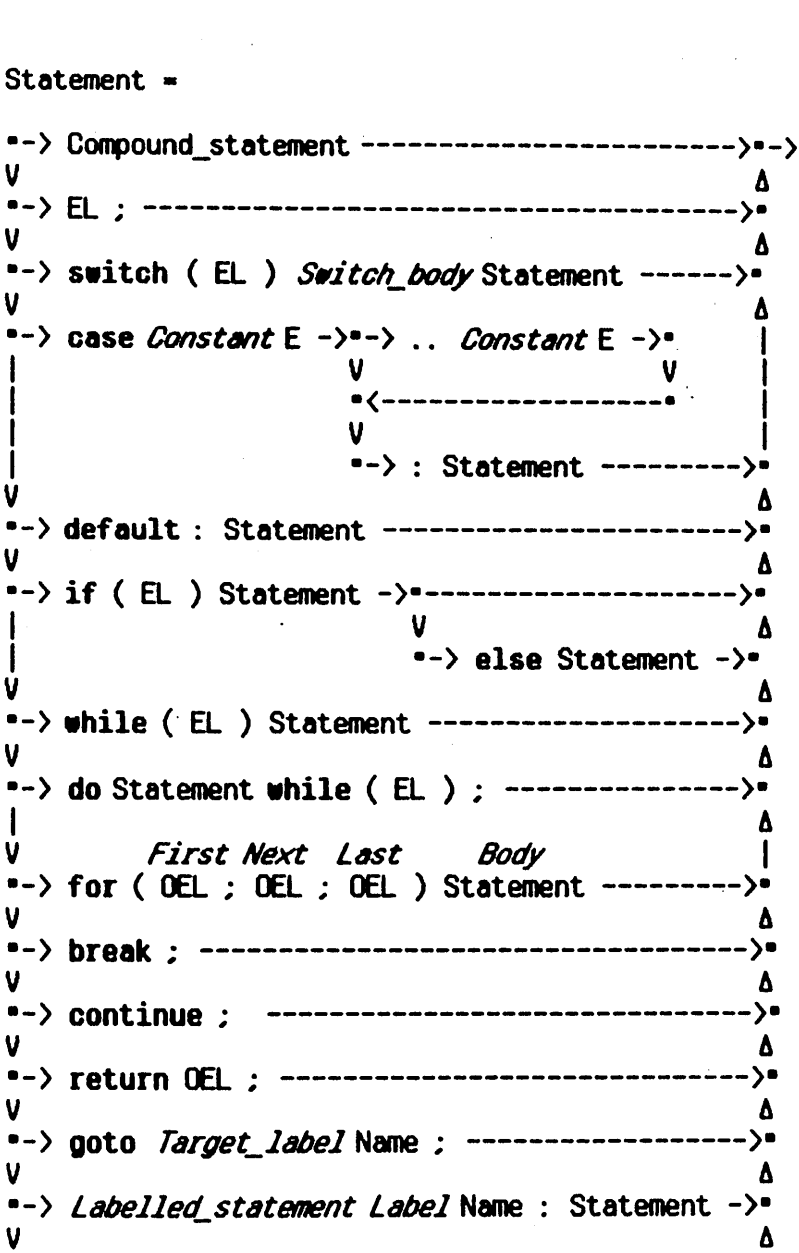
Parameter_types =



Compound_statement =



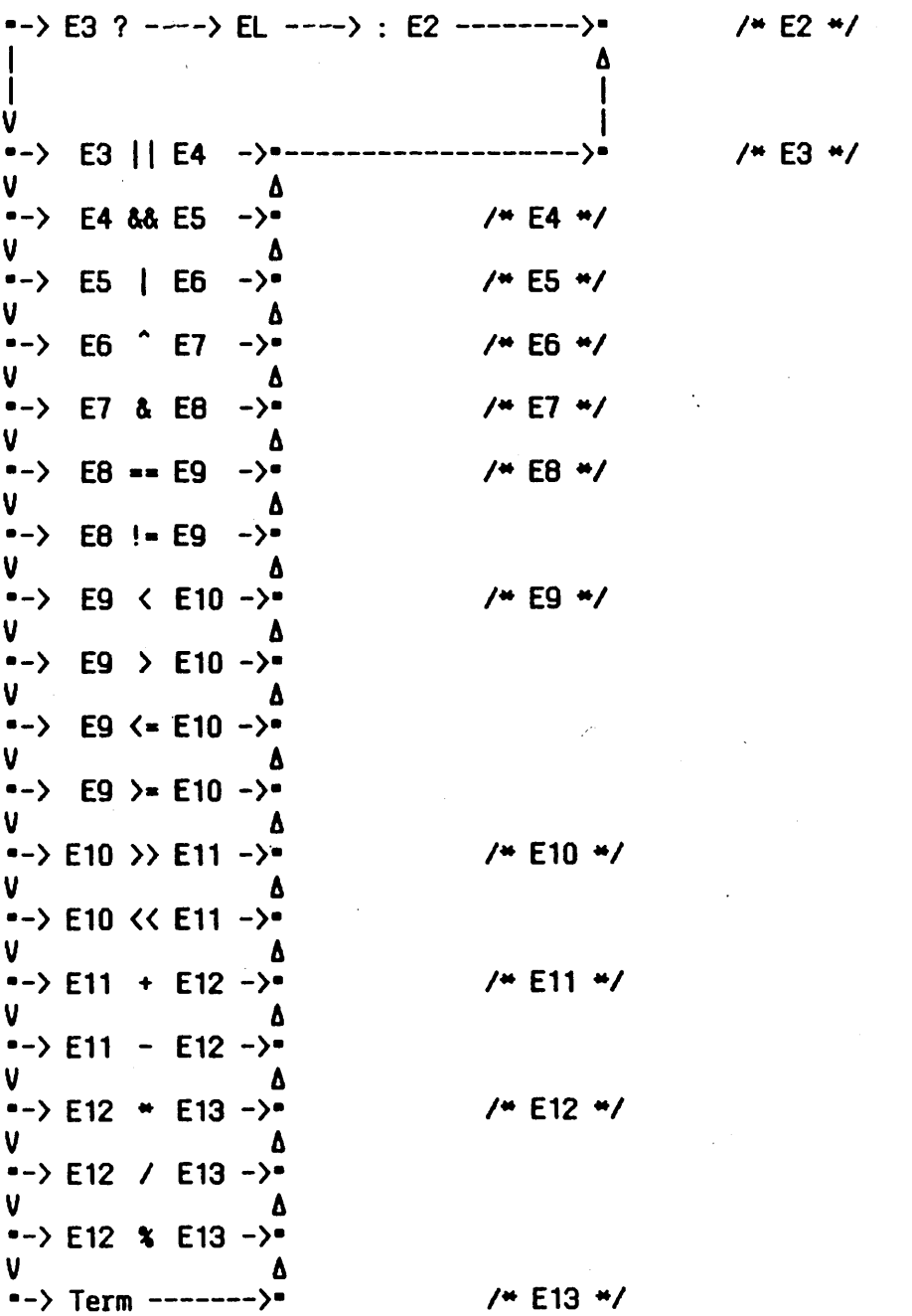
Statement =



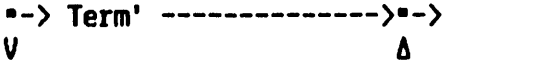
```
OEL =                   /*  Optional Expression List.      */

•-> E ->•->             /*  Used only in Statements.       */
V       Δ
•------->•

EL =                    /*  Equivalently:                  */
                        /*                                 */
•-> E , EL ->•->        /*  •-> E ->•->                    */
V            Δ          /*  Δ       V                      */
•-> E ------>•          /*  •<- , <-•                      */

E =             /* Operator precedence levels are         */
                /* indicated by the "subscripts" on E:    */
                /* larger subscripts mean more binding.   */
```

Assignment_operator:

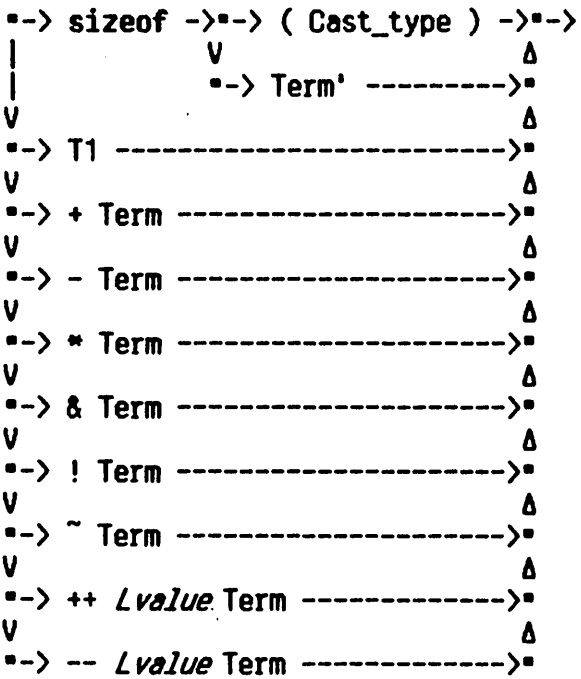
```
•-> Lvalue Term' ->•->  = -->•-> E1 ->•->       /* E1 */
|                    V          Δ        Δ
|                    •-> |= -->•         |
|                    V          Δ        |
|                    •-> ^= -->•         |
|                    V          Δ        |
|                    •-> &= -->•         |
|                    V          Δ        |
|                    •-> >>= ->•         |
|                    V          Δ        |
|                    •-> <<= ->•         |
|                    V          Δ        |
|                    •-> += -->•         |
|                    V          Δ        |
|                    •-> -= -->•         |
|                    V          Δ        |
|                    •-> *= -->•         |
|                    V          Δ        |
|                    •-> /= -->•         |
|                    V          Δ        |
|                    •-> %= -->•         |
|                                        |
V   Conditional_expression:              Δ
```



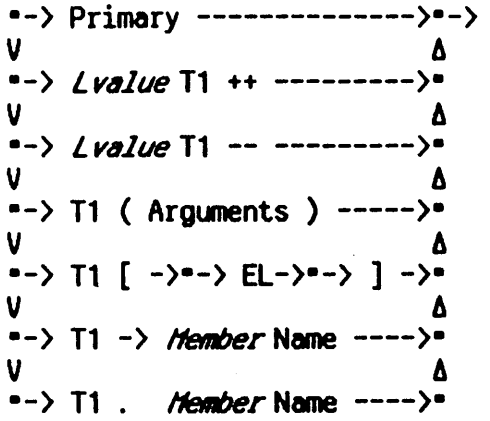
Term =



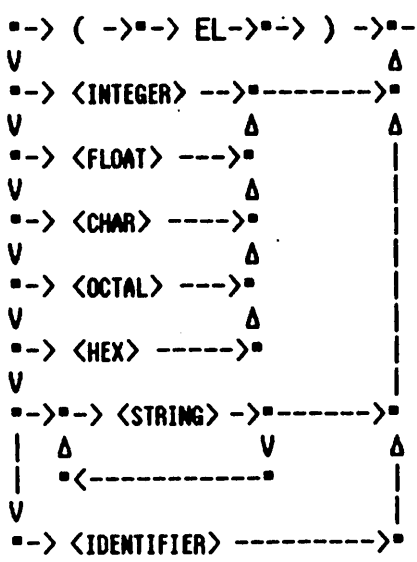
Term' =



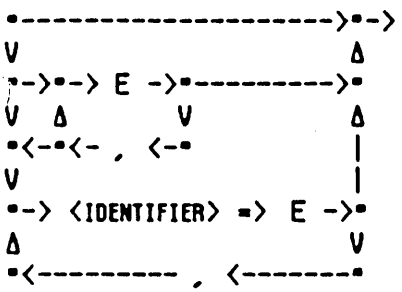
T1 =



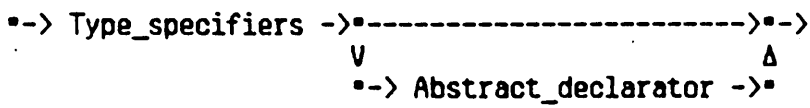
Primary =



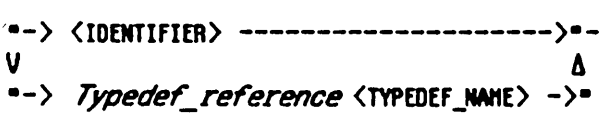
Arguments =



Cast_type =



Name =

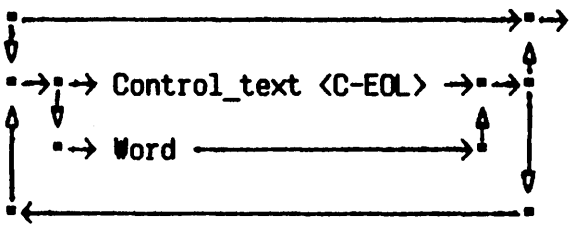


end High_C_phrase_structure

Appendix D
High C ™ Preprocessor Phrase-Structure Chart

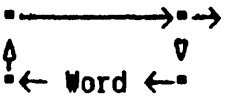
The chart below was produced by the MetaWare Translator Writing System (TWS) from the phrase-structure grammar of Section *Preprocessor,* also found in Appendix *Collected Grammars.* The large diagram was touched up by hand for increased efficiency of space usage. For more information on how to read this chart, consult the introductory paragraphs in Appendix *High C Phrase-Structure Chart.*

C_preprocessor_text =

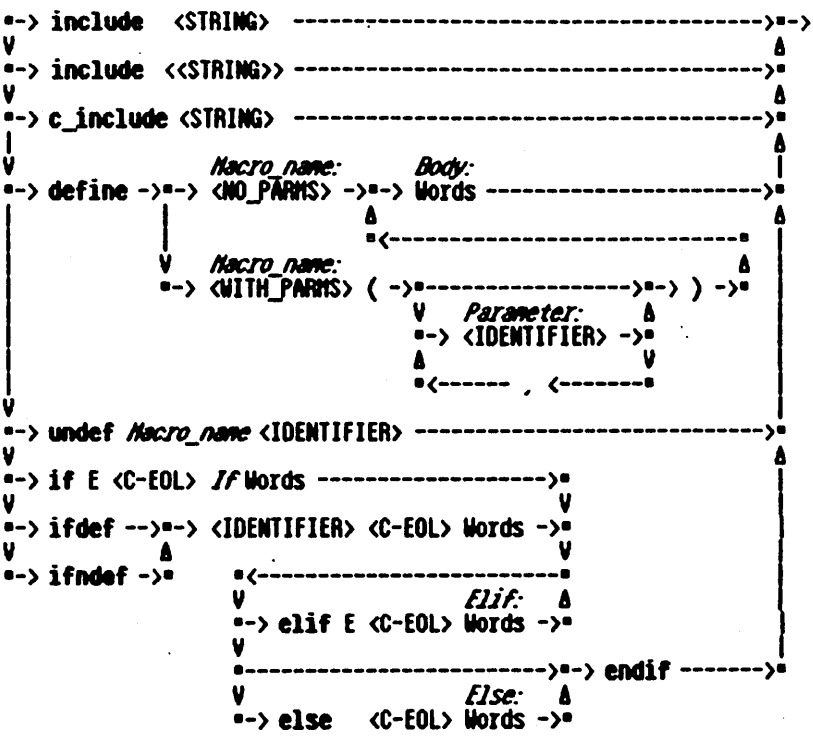



```
Word =          /* Any word from the lexical analyzer except two: */

•→ Any except '<CONTROL>' except '<C-EOL>' →
```

Words =



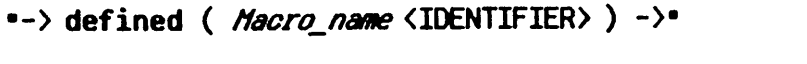
Control_text =



E = /* See the High C PSG. */

•-> C_conditional_compilation_expression ->

/* The E nonterminal generates the same language as E2 in the C phrase- */
/* structure grammar (PSG: see Section *Expressions* and below), except that*/
/* Primary is extended with the following two additional alternatives: */



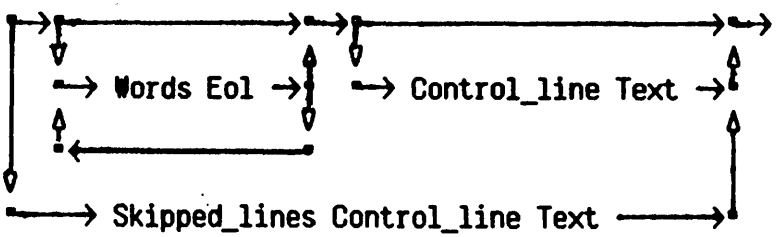
end C_preprocessor_text

Appendix E
High C ™ Lexical Chart

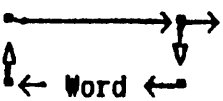
The chart below was produced by the MetaWare Translator Writing System (TWS) from the grammar that also produces tables by which High C compilers lexically analyze their input programs. A few of the diagrams in the chart have been touched up by hand for increased efficiency of space usage. For more information on how to read such charts, consult the introductory paragraphs of Appendix *High C Phrase-Structure Chart*.

C_lexicon =

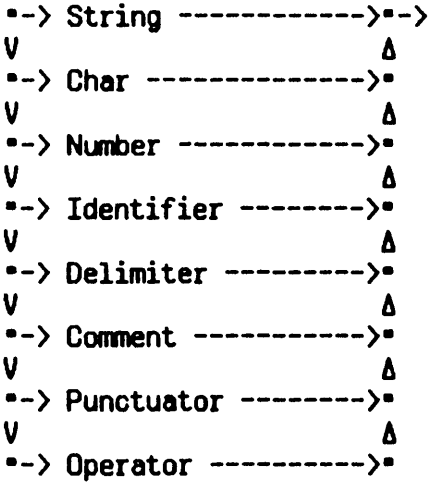
Text =



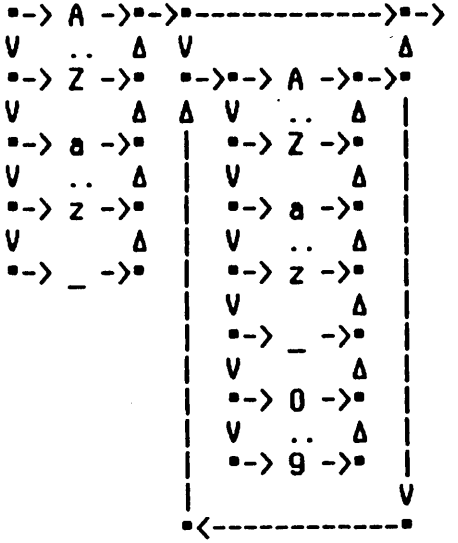
Words =



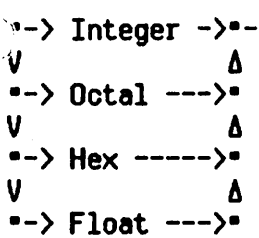
Word =



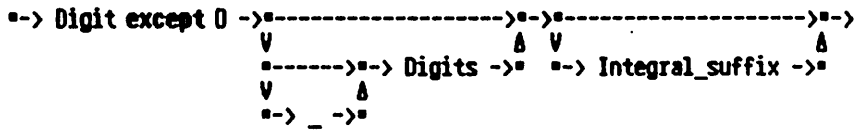
Identifier =
Id_text =



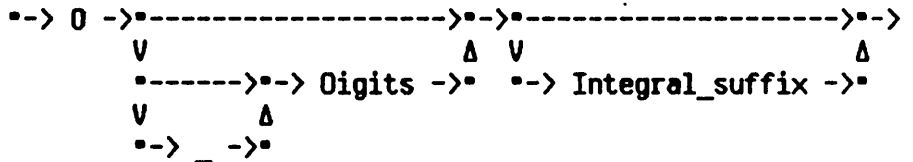
Number =



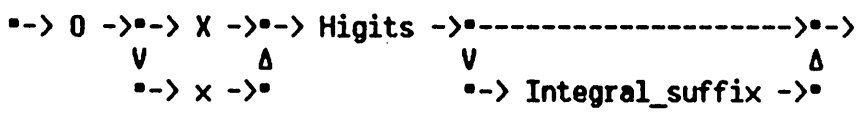
Integer =



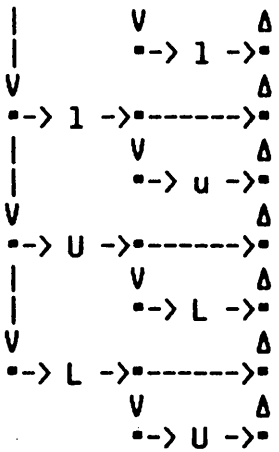
Octal =



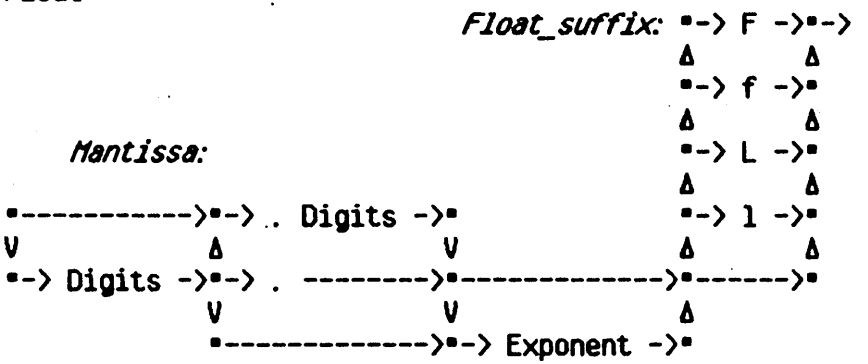
Hex =



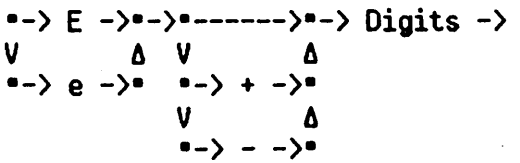
Integral_suffix =



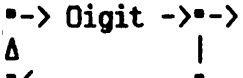
Float =



Exponent =

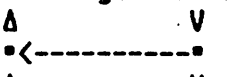


Oigits =

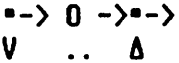


Digits =

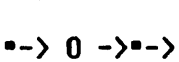
Higits =



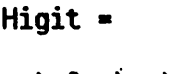
Oigit =



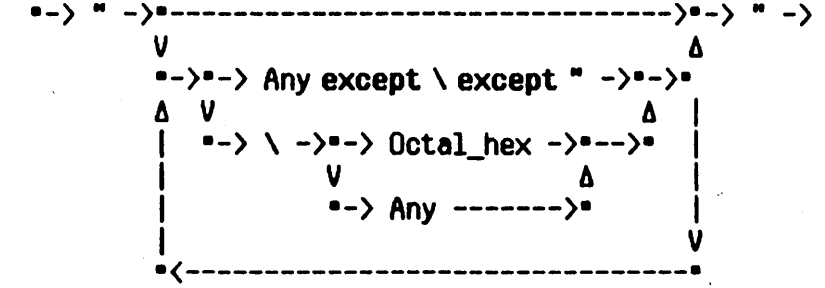
Digit =



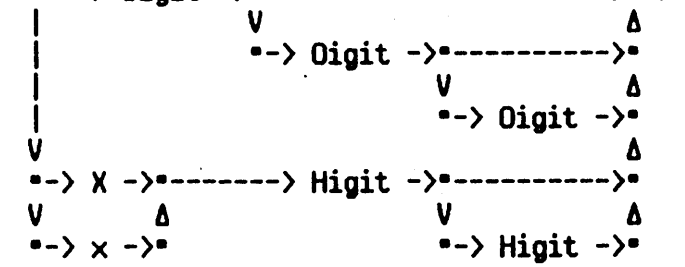
Higit =



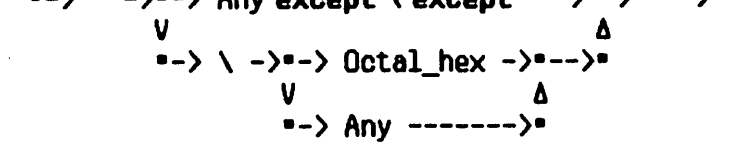
String =
String_text =



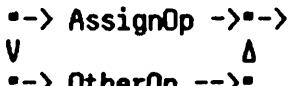
Octal_hex =



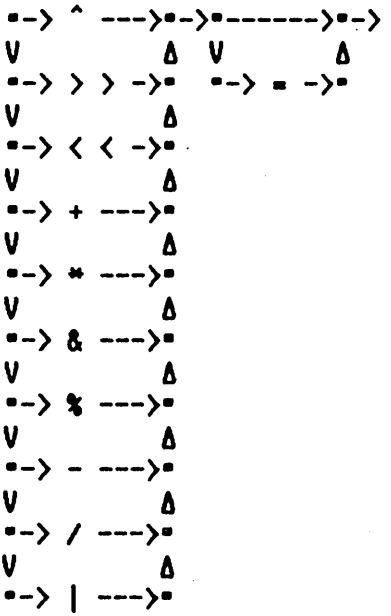
Char =



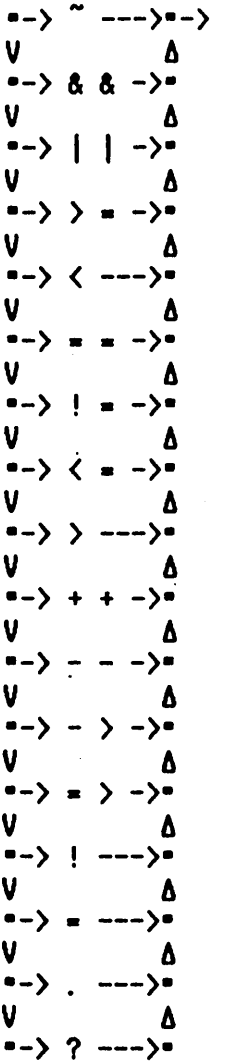
Operator =



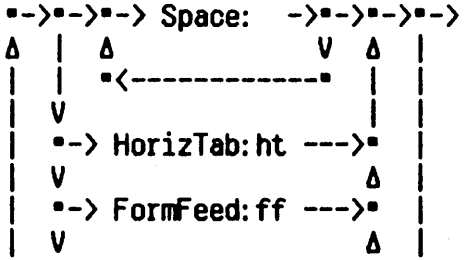
AssignOp =



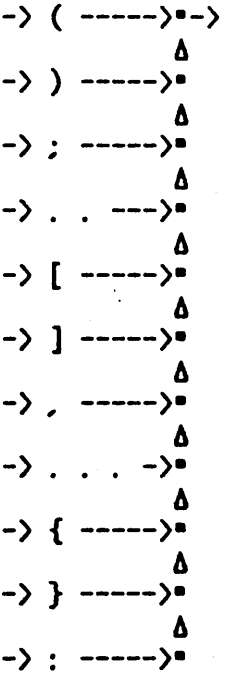
OtherOp =



Delimiter =

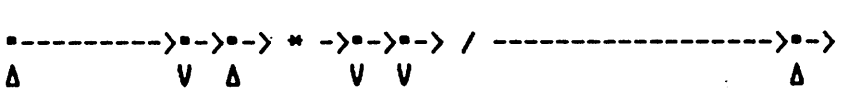


Punctuator =

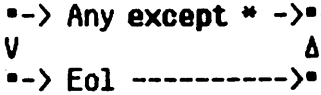


Comment =

Rest =

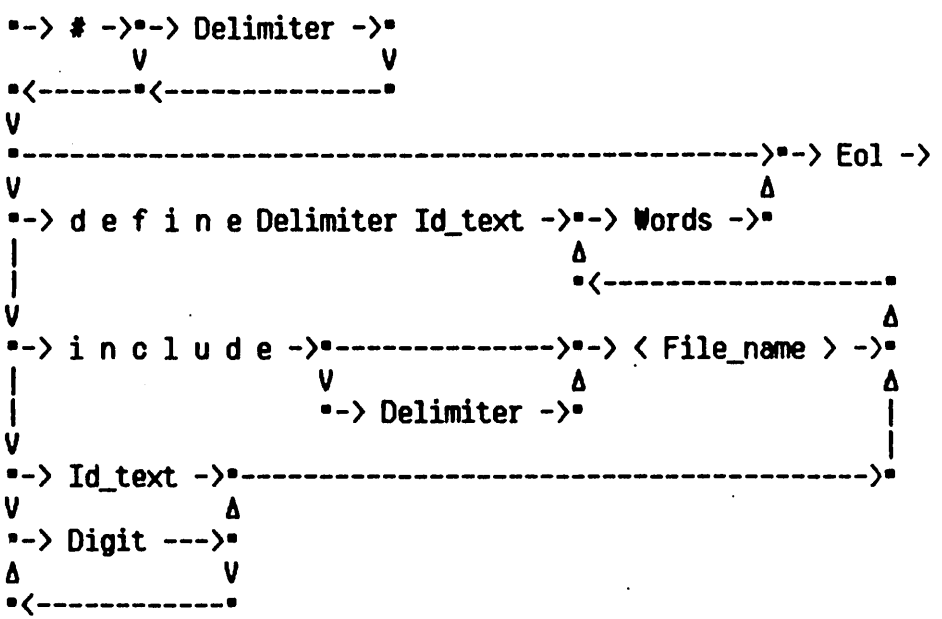


Most =

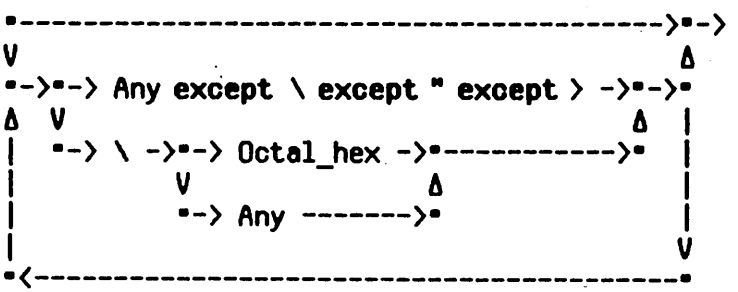


 # Preprocessor lexicon:

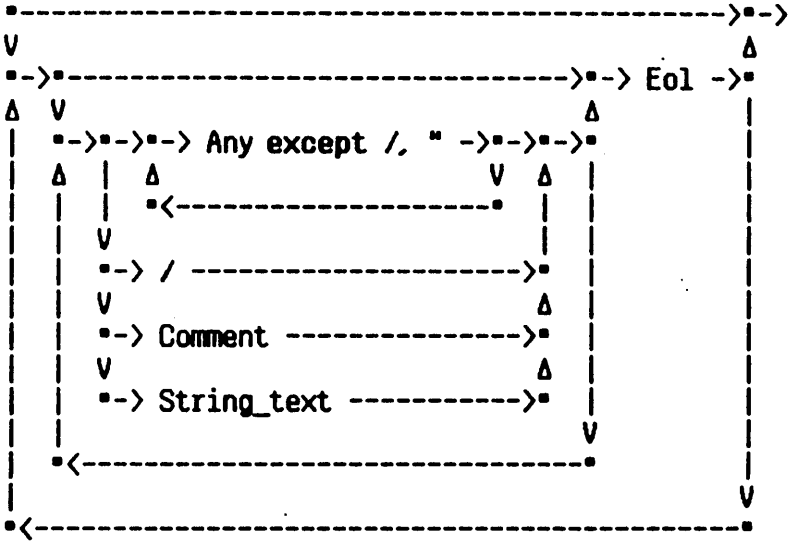
Control_line =



File_name =



Skipped_lines =



end C_lexicon

Index

Starting on the next page is a "permuted key word in context" index for this document. In the center column is the particular key word W being indexed, in the context of a phrase or sentence containing W. The phrase appears to the left and right of W.

Occasionally the text of the phrase preceding W does not fit in the space to the left of W. In that case the index entry looks like

```
is text that was too long to precede the  WORD being indexed.  This ..... 7.4
```

where the first word "This" of the sentence did not fit on the left. Similarly the text to the right of W can be crowded:

```
                    the right.  This  WORD is followed by too much text on ..... 7.4
```

where "the right" did not fit on the right.

If the texts both to the left and right do not fit, or the left (right) text cannot be completely wrapped around to the right (left), the entry is continued on another line. For example:

```
   but not too much text on the left.  This  WORD is followed by far too much text on...
...the right                                                                  ..... 7.4
```

After locating an entry, proceed directly to the referenced section(s). If a reference is to Section X.Y, look on page X-Y first and you will usually be within a page of the desired referent.

More Feedback, Please

(After some use.)

We would greatly appreciate your ideas regarding improvement of the language, its compiler, and its documentation. Please take time to jot down your ideas on this page (front and back) and on additional sheets as necessary ***as you use the software***. Then, after you have some significant experience with the software, please mail the results to:

MetaWare™ Incorporated
412 Liberty Street
Santa Cruz, CA 95060

MetaWare may use or distribute any information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use that information. If you wish a reply, please provide your name and address. **Thank you in advance, The Authors.**

Page Comment

More Feedback, Please

Page Comment

Acknowledgments

The authors of these manuals and designers of the High C language would like to thank the C standards committee, whose drafts of the C standard helped illuminate many dark areas of the language and assisted greatly in "chunking" the language concepts.

Paul Redmond's feedback was invaluable as he put dBase III through High C for Ashton-Tate. In the process he helped us polish the compiler in many ways.

David Shields' efforts in working with us were also very beneficial. He put tens of thousands of lines of C source code through High C, transliterated from the SETL version of the Ada-Ed compiler at New York University.

Professor William McKeeman and his research group at the Wang Institute of Graduate Studies supplied us with a collection of "gray expressions" that helped us verify the compiler.

The support of others who must needs remain nameless at this time is also appreciated.

Most of all we acknowledge that we are not self-made, but God-made. And we thank God for building into us the talents that made it possible for us to create High C. Praise God, from whom all blessings flow.

Ad majorem Dei gloriam (A.M.D.G.).

This ends the

High C ™

Language Reference Manual