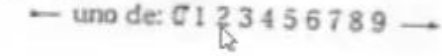
que indica que debe escogerse una de las alternativas. Para abreviar algunas de las selecciones con muchas opciones, utilizamos la frase *uno de:*.

C.1 Elementos Básicos

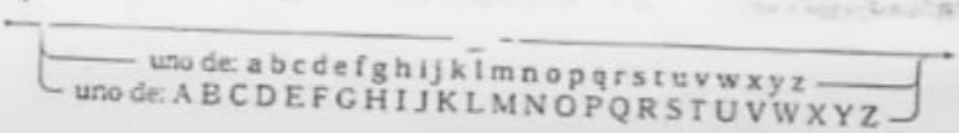
⟨*dígito_decimal*⟩ =



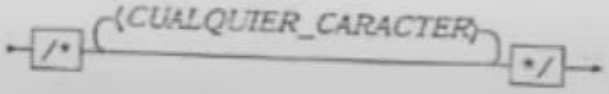
⟨*dígito_octal*⟩ =

⟨*dígito_hexadecimal*⟩ =

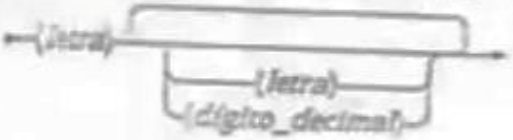
⟨*letra*⟩ =



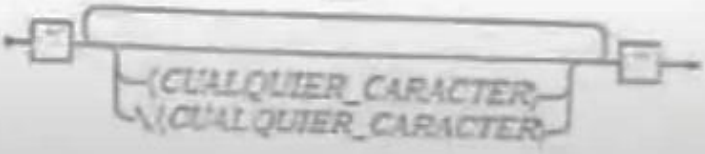
⟨*comentario*⟩ =



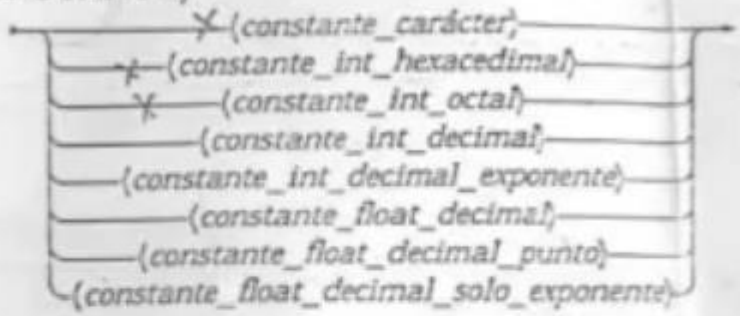
⟨*IDENTIFICADOR_o_NOMBRE_DE_TIPO*⟩ =



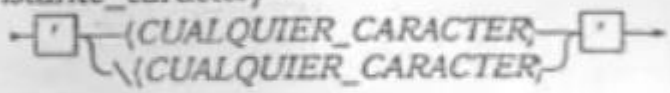
⟨*STRING_CONSTANTE*⟩ =



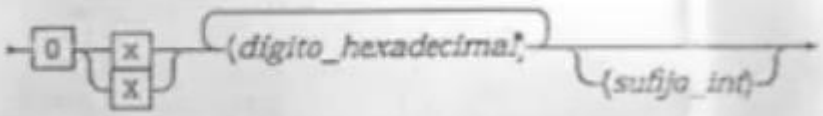
⟨CONSTANTE⟩ =



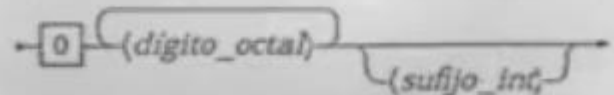
⟨constante_carácter⟩ =



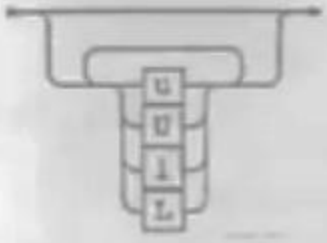
⟨constante_int_hexadecimal⟩ =



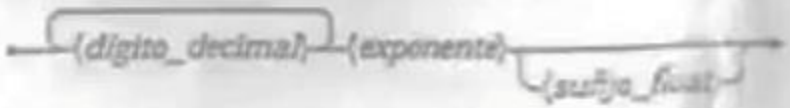
⟨constante_int_octal⟩ =



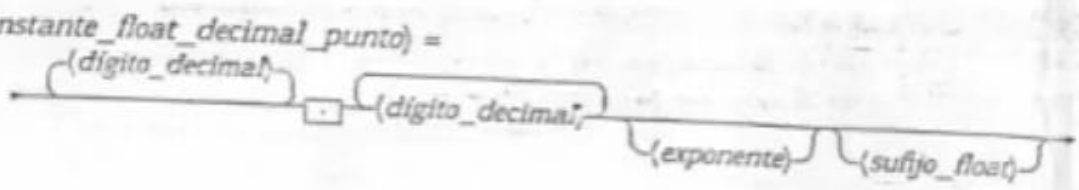
⟨sufijo_int⟩ =



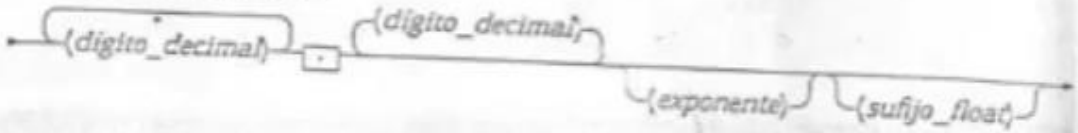
⟨constante_float_solo_exponente⟩ =



⟨constante_float_decimal_punto⟩ =

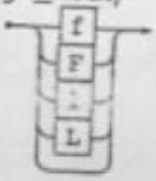


⟨constante_float_decimal⟩ =



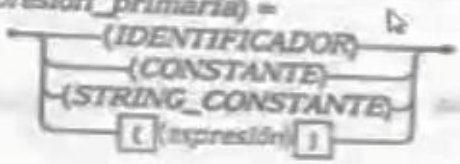
⟨exponente⟩ =

⟨sufijo_float⟩ =

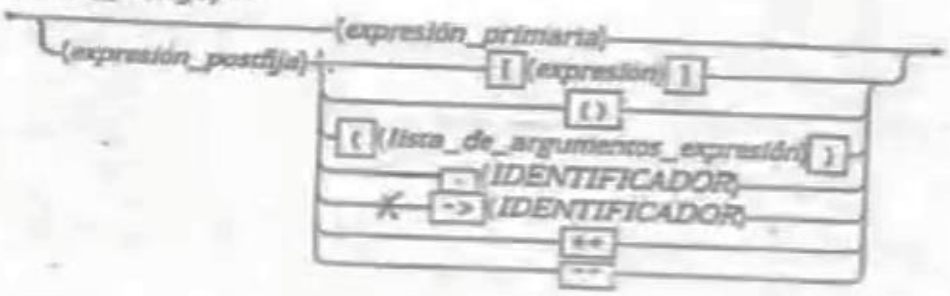


C.2 Expresiones

⟨expresión_primaria⟩ =

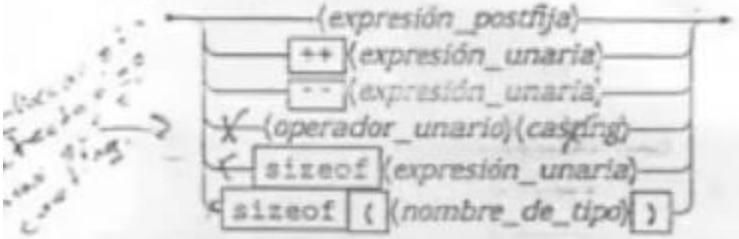


⟨expresión_postfija⟩ =



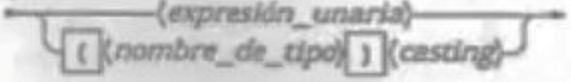
⟨lista_de_argumentos_expresión⟩ =

⟨expresión_unaria⟩ =

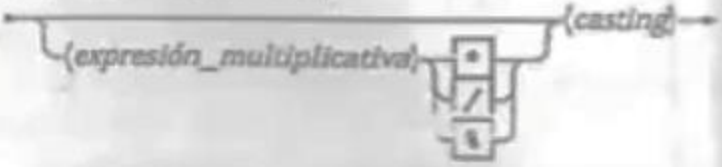


⟨operador_unario⟩ =

⟨casting⟩ =

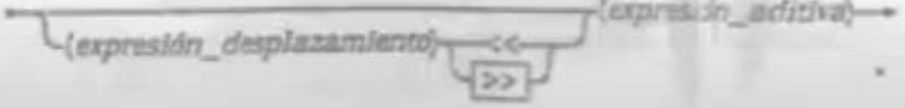


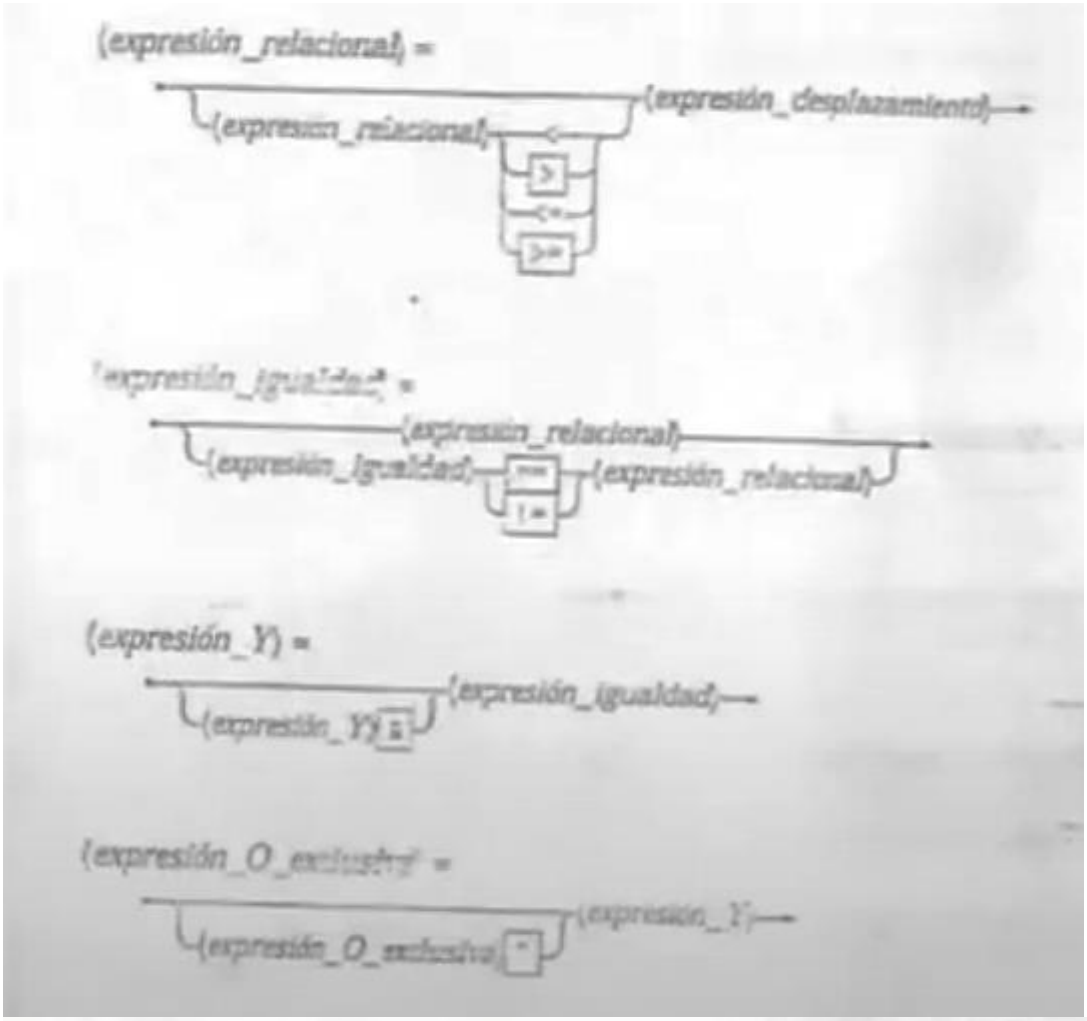
⟨expresión_multiplicativa⟩ =

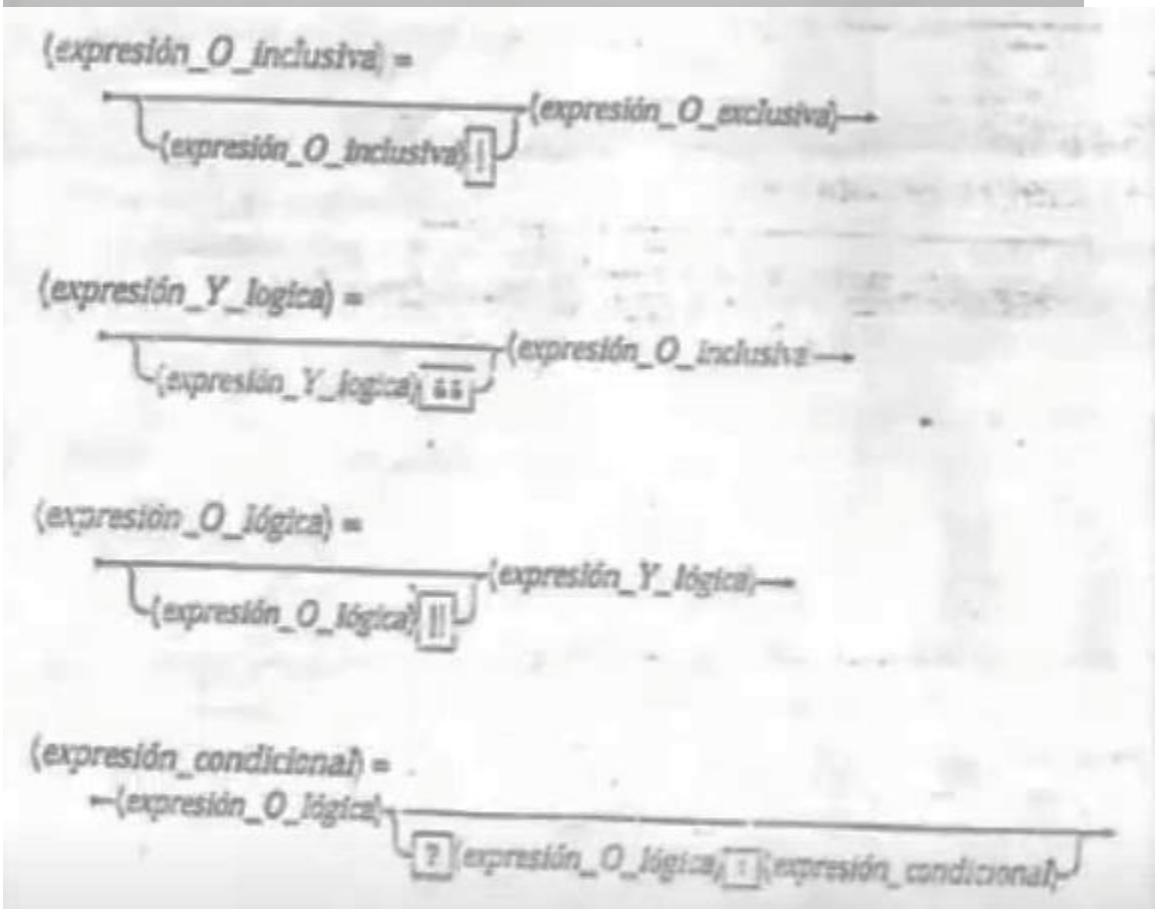


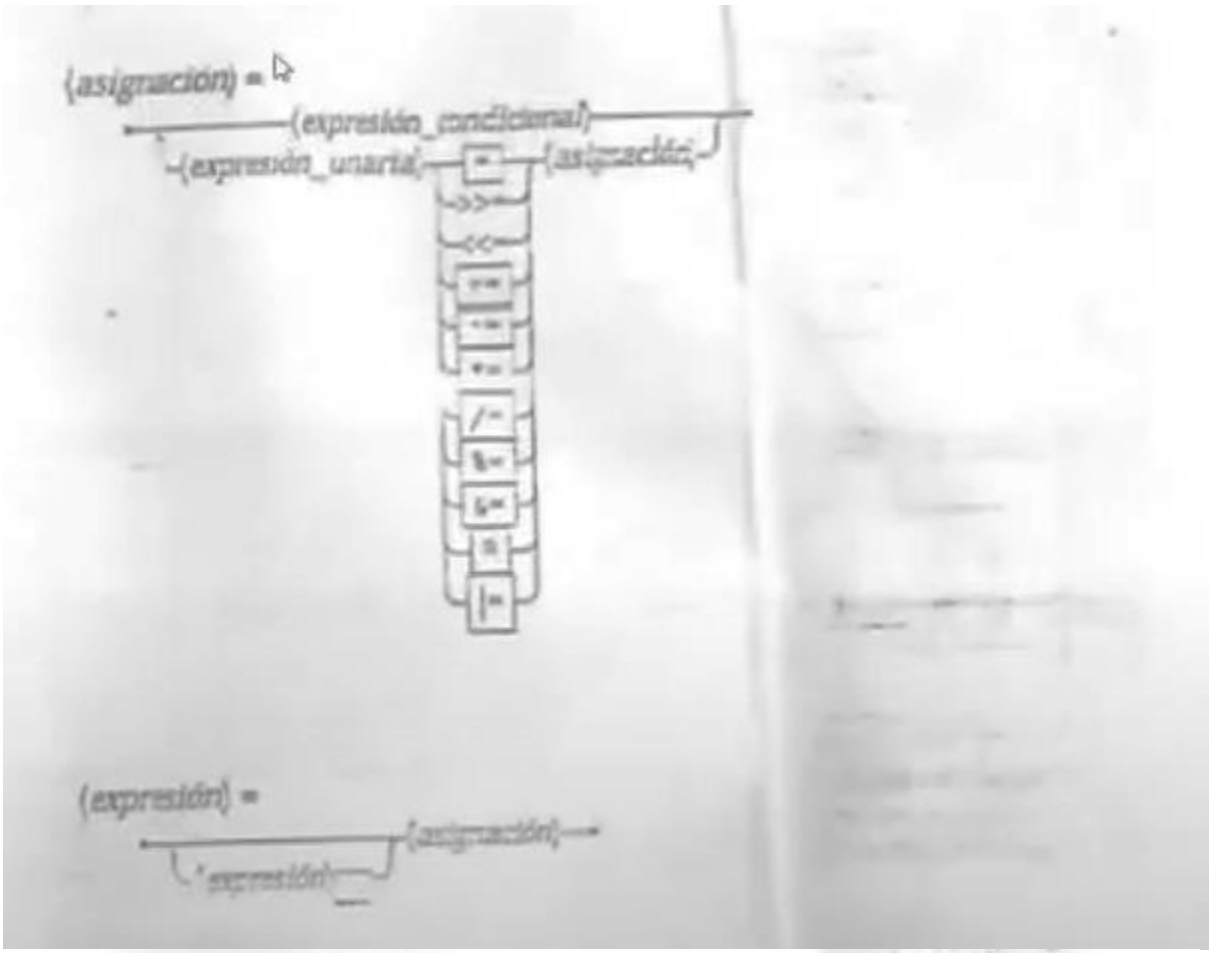
⟨expresión_aditiva⟩ =

⟨expresión_desplazamiento⟩ =




⟨expresión_relacional⟩ =
⟨expresión_relacional⟩
>
<=
>=
⟨expresión_desplazamiento⟩
⟨expresión_igualdad⟩ =
⟨expresión_igualdad⟩
==
!=
⟨expresión_relacional⟩
⟨expresión_relacional⟩
⟨expresión_Y⟩ =
⟨expresión_Y⟩
&
⟨expresión_igualdad⟩
⟨expresión_O_exclusiva⟩ =
⟨expresión_O_exclusiva⟩
^
⟨expresión_Y⟩


⟨expresión_O_inclusiva⟩ =
⟨expresión_O_inclusiva⟩
|
⟨expresión_O_exclusiva⟩
⟨expresión_Y_lógica⟩ =
⟨expresión_Y_lógica⟩
&&
⟨expresión_O_inclusiva⟩
⟨expresión_O_lógica⟩ =
⟨expresión_O_lógica⟩
||
⟨expresión_Y_lógica⟩
⟨expresión_condicional⟩ =
⟨expresión_O_lógica⟩
?
⟨expresión_O_lógica⟩
:
⟨expresión_condicional⟩



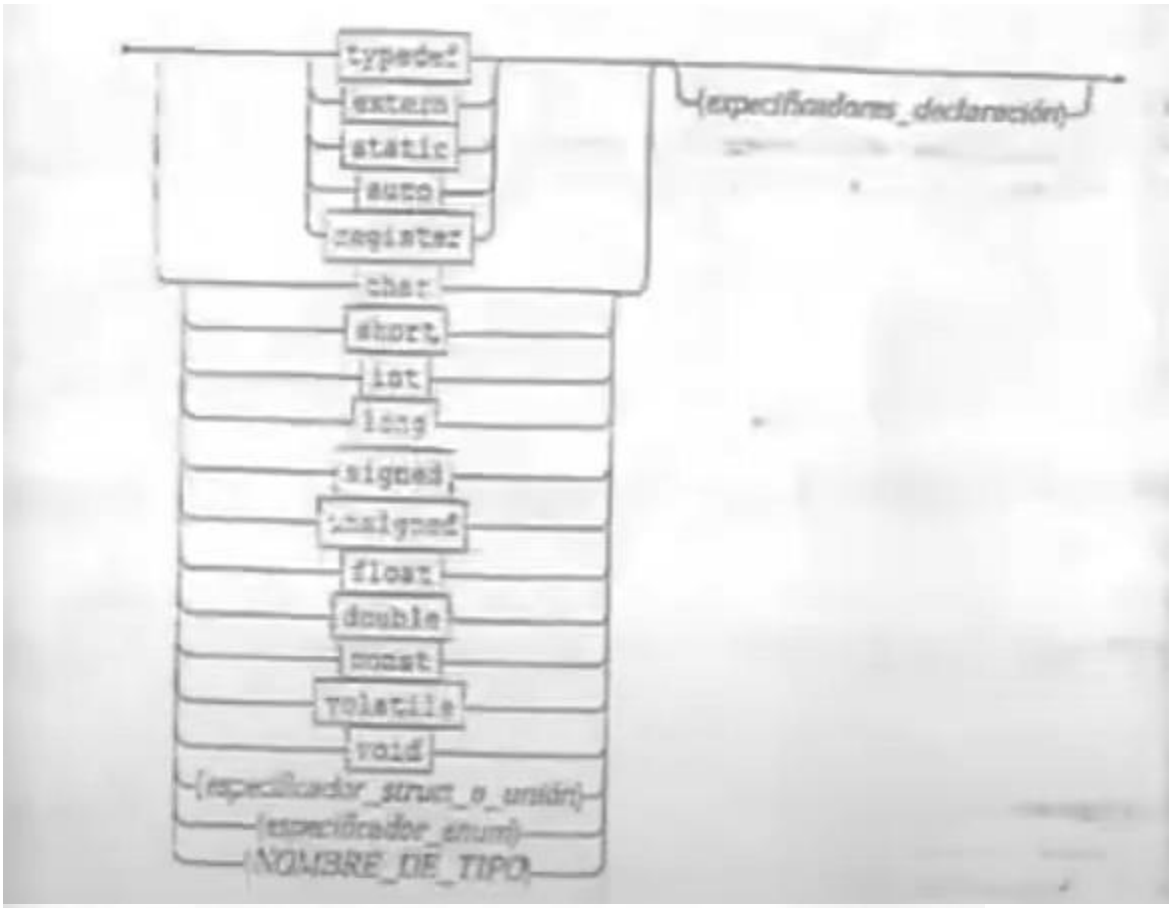
⟨expresión_constante⟩ =
⟨expresión_condicional⟩

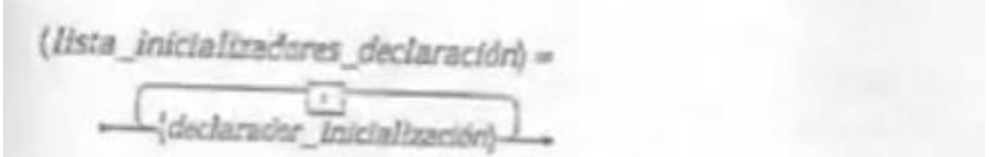
C.3 Declaraciones

(1) ⟨declaración⟩ =
⟨especificadores_declaración⟩
⟨lista_inicializadores_declaración⟩
;

(2) ⟨especificadores_declaración⟩ =
⟨especificador_de_almacenamiento⟩
⟨especificador_de_tipo⟩
⟨especificadores_declaración⟩

(3) ⟨especificadores_declaración⟩ =

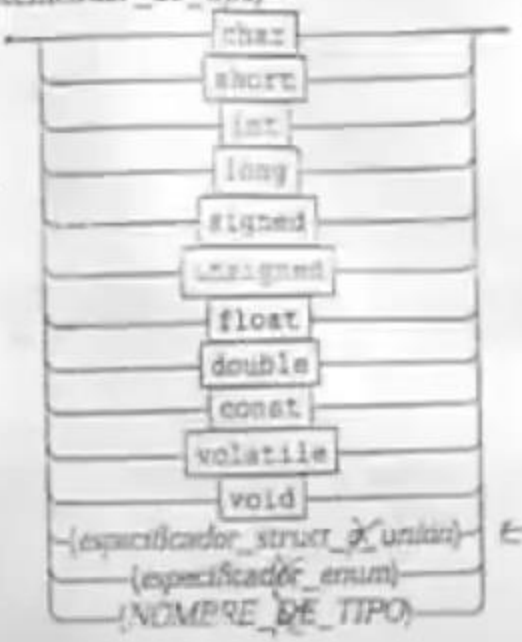

typedef
extern
static
auto
register
char
short
int
long
signed
unsigned
float
double
const
volatile
void
(especificador_struct_o_unión)
(especificador_enum)
NOMBRE_DE_TIPO
(especificadores_declaración)


(lista_inicializadores_declaración) =
,
(declarador_inicialización)


(declarador_inicialización) =
(declarador)
=
(inicializador)

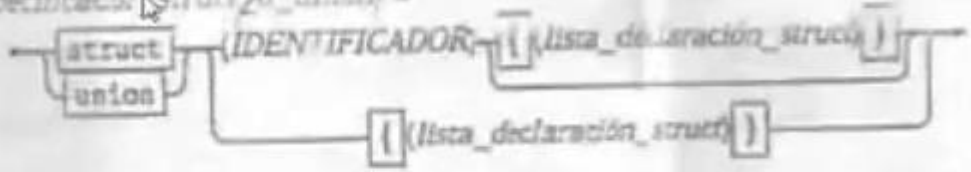

(especificador_de_almacenamiento) =
typedef
extern
static
auto
register

(7) (especificador_de_tipo) =

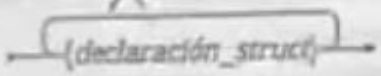


← especificador_de_clase

(8) (especificador_struct_o_unión) =

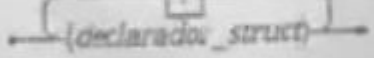


(9) (lista_declaración_struct) =

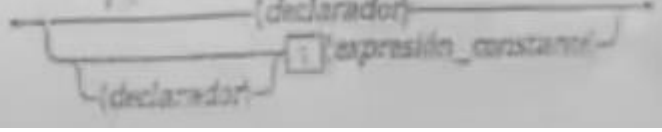


(10) (declaración_struct) =

(11) (lista_declarador_struct) =



(12) (declarador_struct) =



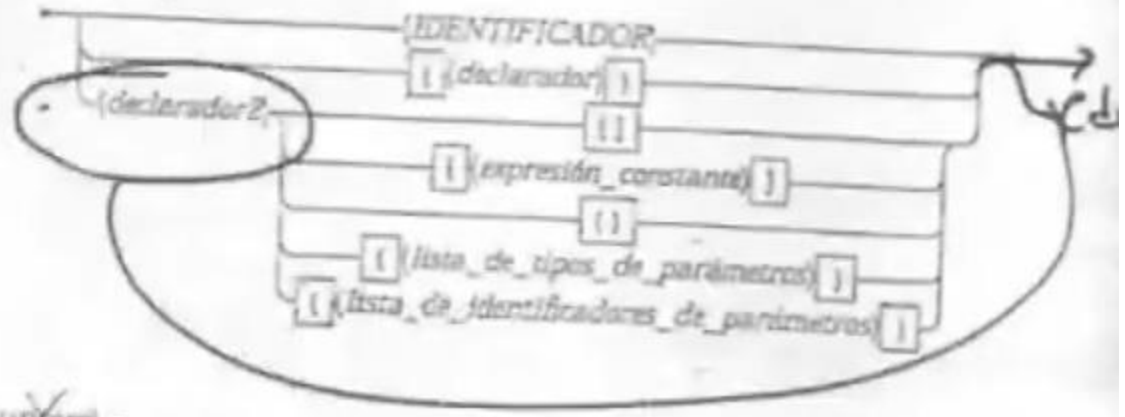
(13) ⟨especificador_enum⟩ =

(14) ⟨lista_enumeradora⟩ =

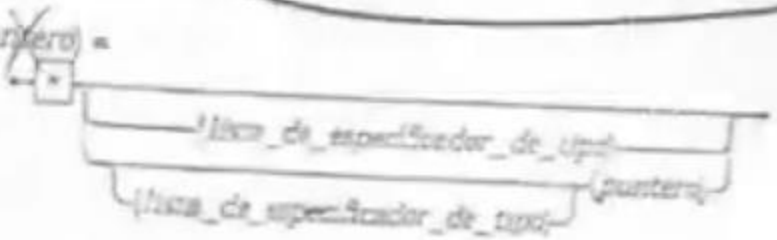
(15) ⟨enumerador⟩ =

(16) ⟨declarador⟩ =

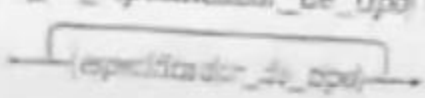
⟨declarador2⟩ =



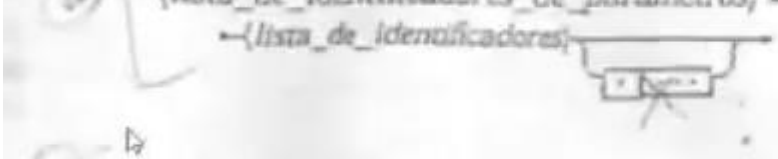
⟨puntero⟩ =



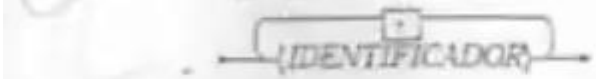
⟨lista_de_especificador_de_tipo⟩ =



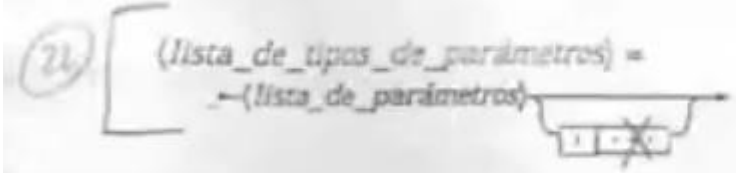
(20) (lista_de_identificadores_de_parámetros) =



(22) (lista_de_identificadores) =

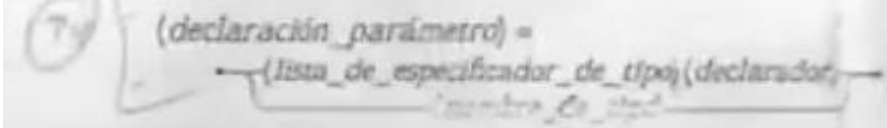


(24) (lista_de_tipos_de_parámetros) =

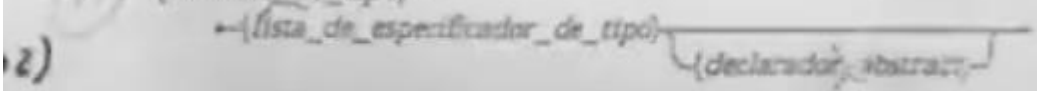


(25) (lista_de_parámetros) =

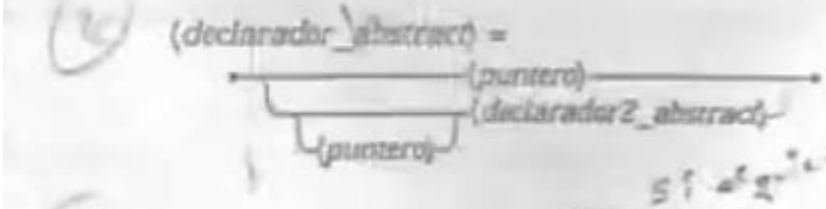
(27) (declaración_parámetro) =



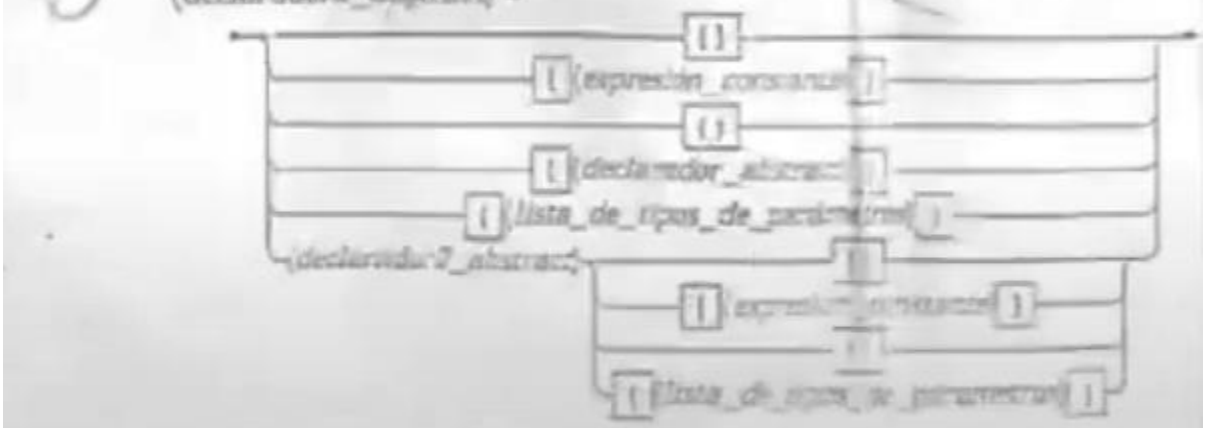
(28) (nombre_de_tipo) =



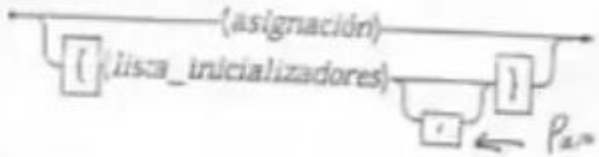
(32) (declarador_abstract) =



(33) (declarador2_abstract) =

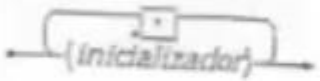


28 ⟨inicializador⟩ =



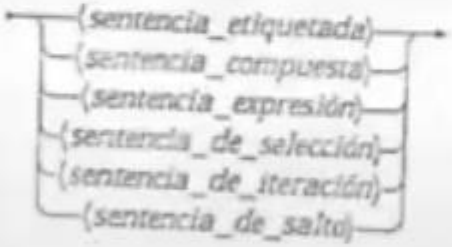
← Para int ?

29 ⟨lista_inicializadores⟩ =

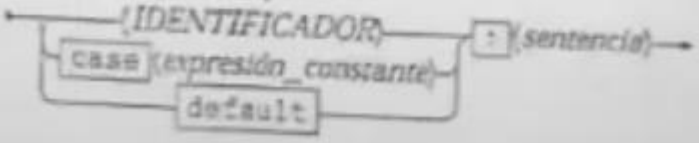


C.4 Sentencias

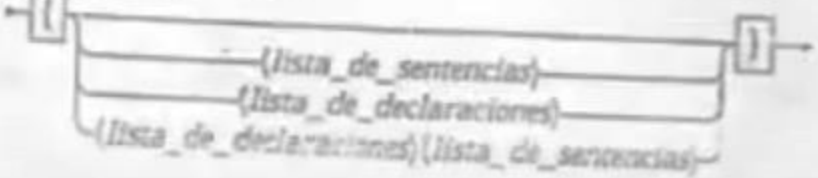
⟨sentencia⟩ =



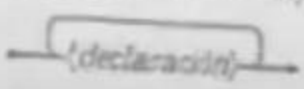
⟨sentencia_etiquetada⟩ =



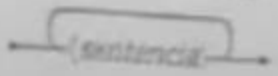
⟨sentencia_compuesta⟩ =



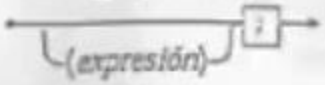
⟨lista_de_declaraciones⟩ =



⟨lista_de_sentencias⟩ =

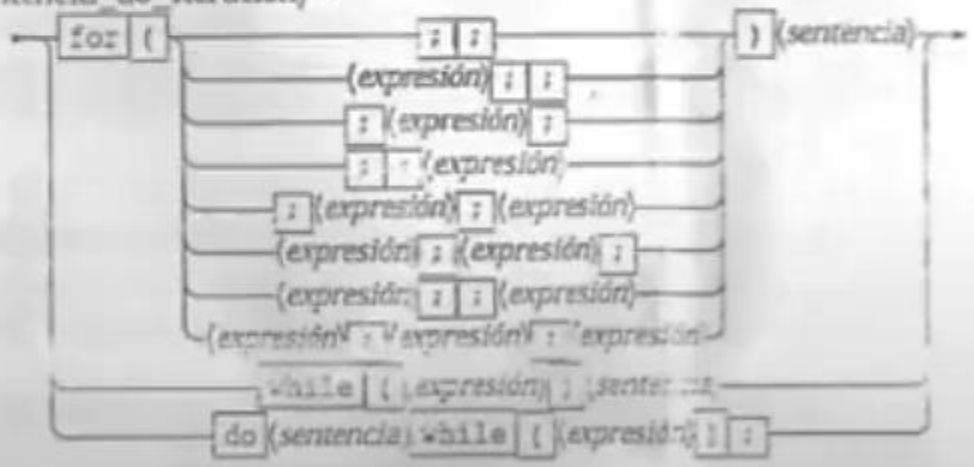


⟨sentencia_expresión⟩ =

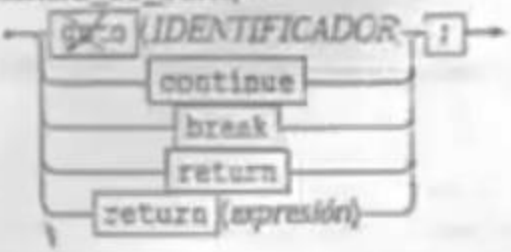


⟨sentencia_de_selección⟩ =

⟨sentencia_de_iteración⟩ =

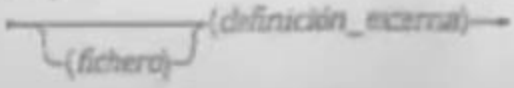


⟨sentencia_de_salto⟩ =

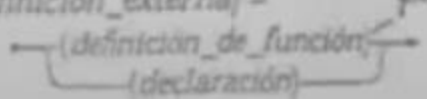


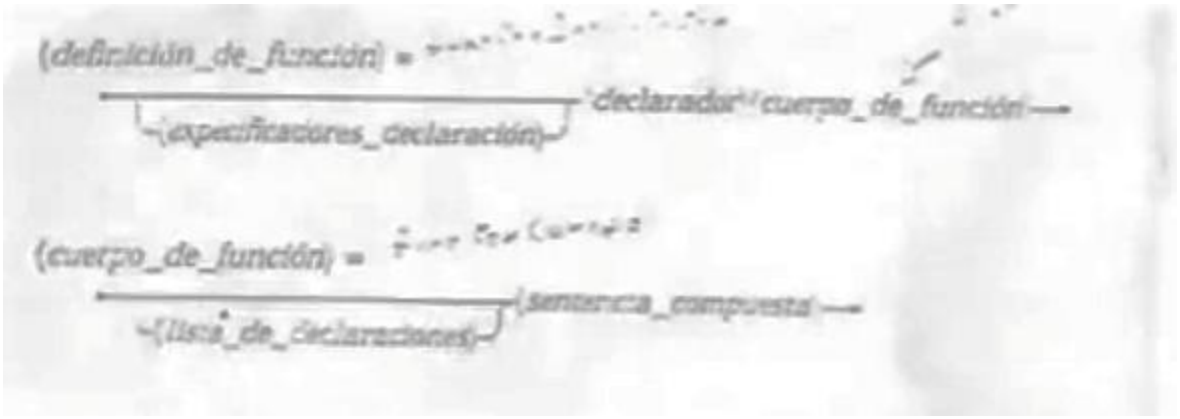
C.5 Estructura del Fichero

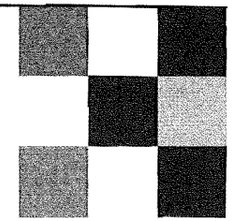
⟨fichero⟩ =



⟨definición_externa⟩ =




⟨definición_de_función⟩ →
⟨especificadores_declaración⟩
'declarador' ⟨cuerpo_de_función⟩ →
⟨cuerpo_de_función⟩ →
⟨lista_de_declaraciones⟩
⟨sentencia_compuesta⟩ →



Appendix B

Language Syntax

In this appendix, we give an extended BNF syntax for the ANSI version of the C language. (See Section 2.2, "Syntax Rules," on page 73.) This syntax, although intended for the human reader, is concisely written. The C language is inherently context-sensitive; restrictions and special cases are left to the main text. The conceptual output of the preprocessor is called a *translation unit*. The syntax of the C language pertains to translation units. The syntax for preprocessing directives is independent of the rest of the C language. We present it at the end of this appendix.

B.1 Program

program ::= { *file* }$_{1+}$

file ::= *decls_and_fct_definitions*

decls_and_fct_definitions ::= { *declaration* }$_{1+}$ *decls_and_fct_definitions*$_{\text{opt}}$
| { *function_definition* }$_{1+}$*decls_and_fct_definitions*$_{\text{opt}}$

B.2 Function Definition

function_definition ::= { `extern` | `static` }$_{opt}$ *type_specifier*
function_name (*parameter_declaration_list*$_{opt}$)
compound_statement

function_name ::= *identifier*

parameter_declaration_list ::= *parameter_declaration* { , *parameter_declaration* }$_{0+}$

B.3 Declaration

declaration ::= *declaration_specifiers* *init_declarator_list*$_{opt}$

declaration_specifiers ::= *storage_class_specifier_or_typedef* *declaration_specifiers*$_{opt}$
| *type_specifier* *declaration_specifiers*$_{opt}$
| *type_qualifier* *declaration_specifiers*$_{opt}$

storage_class_specifier_or_typedef ::= `auto` | `extern` | `register` | `static`
| `typedef`

type_specifier ::= `char` | `double` | `float` | `int` | `long` | `short` | `signed`
| `unsigned`
| `void` | *enum_specifier* | *struct_or_union_specifier*
| *typedef_name*

enum_specifier ::= `enum` *tag*$_{opt}$ { *enumerator_list* } | `enum` *tag*

tag ::= *identifier*

enumerator_list ::= *enumerator* { , *enumerator* }$_{opt}$

enumerator ::= *enumeration_constant* { = *const_integral_expr* }$_{opt}$

enumeration_constant ::= *identifier*

struct_or_union_specifier ::= *struct_or_union* *tag*$_{opt}$ { *struct_declaration_list* }
| *struct_or_union* *tag*

struct_or_union ::= `struct` | `union`

struct_declaration_list ::= { *struct_declaration* }$_{1+}$

struct_declaration ::= *type_specifier_qualifier_list* *struct_declarator_list* ;

type_specifier_qualifier_list ::= *type_specifier* *type_specifier_qualifier_list*$_{opt}$
| *type_qualifier* *type_specifier_qualifier_list*$_{opt}$

struct_declarator_list ::= *struct_declarator* { , *struct_declarator* }$_{0+}$

struct_declarator ::= *declarator* | *declarator*$_{opt}$: *const_integral_expr*

type_qualifier ::= `const` | `volatile`

declarator ::= *pointer*$_{opt}$ *direct_declarator*

pointer ::= { * | *type_qualifier_list*$_{opt}$ }$_{1+}$

type_qualifier_list ::= { *type_qualifier* }$_{1+}$

direct_declarator ::= *identifier* | (*declarator*)
| *direct_declarator* [*const_integral_expr*$_{opt}$]
| *direct_declarator* (*parameter_type_list*)
| *direct_declarator* (identifier_list$_{opt}$)

parameter_type_list ::= *parameter_list* | *parameter_list* , ...

parameter_list ::= *parameter_declaration* { , *parameter_declaration* }$_{0+}$

parameter_declaration ::= *declaration_specifiers* *declarator*
| *declaration_specifiers* *abstract_declarator*$_{opt}$

abstract_declarator ::= *pointer* | pointer$_{opt}$ *direct_abstract_declarator*

direct_abstract_declarator ::= (*abstract_declarator*)
| *direct_abstract_declarator*$_{opt}$ [*const_integral_expr*$_{opt}$]
| *direct_abstract_declarator*$_{opt}$ (*parameter_type_list*$_{opt}$)

identifier_list ::= *identifier* { , *identifier* }$_{0+}$

typedef_name ::= *identifier*

init_declarator_list ::= *init_declarator* { , *init_declarator* }$_{opt}$

init_declarator ::= *declarator* | *declarator* = *initializer*

initializer ::= *assignment_expression* | { *initializer_list* } | { *initializer_list* , }

initializer_list ::= *initializer* { , *initializer* }$_{0+}$

B.4 Statement

statement ::= *compound_statement* | *expression_statement* | *iteration_statement*
| *jump_statement* | *labeled_statement* | *selection_statement*

compound_statement ::= { *declaration_list*$_{opt}$ *statement_list*$_{opt}$ }

declaration_list ::= { *declaration* }$_{1+}$

statement_list ::= { *statement* }$_{1+}$

expression_statement ::= *expression*$_{opt}$;

jump_statement ::= `break ;` | `continue ;` | `goto` *identifier* ;
| `return` *expression*$_{opt}$;

labeled_statement ::= *identifier* : *statement*
| `case` *const_integral_expr* : *statement*
| `default` : *statement*

selection_statement ::= `if (` *expression* `)` *statement*
| `if (` *expression* `)` *statement* `else` *statement*
| *switch_statement*

switch_statement ::= `switch (` *integral_expression* `)`
{ *case_statement* | `default` : *statement* | *switch_block* }$_1$

case_statement ::= { `case` *const_integral_expr* : }$_{1+}$ *statement*

switch_block ::= { { *declaration_list* }$_{opt}$ *case_default_group* }

case_default_group ::= { *case_group* }$_{1+}$
| { *case_group* }$_{0+}$ *default_group* { *case_group* }$_{0+}$

case_group ::= { `case` *const_integral_expr* : }$_{1+}$ { *statement* }$_{1+}$

default_group ::= `default` : {*statement*}$_{1+}$

B.5 Expression

expression ::= *constant* | *string_literal* | (*expression*) | *lvalue*
| *assignment_expression* | *expression* , *expression* | + *expression*
| - *expression* | *function_expression* | *relational_expression*
| *equality_expression* | *logical_expression*
| *expression arithmetic_op expression* | *bitwise_expression*
| *expression* ? *expression* : *expression* | `sizeof` *expression*
| `sizeof (` *type_name*) | (*type_name*) *expression*

lvalue ::= & *lvalue* | ++ *lvalue* | *lvalue* ++ | -- *lvalue* | *lvalue* --
| *identifier* | * *expression* | *lvalue* [*expression*] | (*lvalue*)
| *lvalue* . *identifier* | *lvalue* -> *identifier*

assignment_expression ::= *lvalue assignment_op expression*

assignment_op ::= = | += | -= | *= | /= | %= | &= | ^= | |= | >>= | <<=

arithmetic_op ::= + | - | * | / | %

relational_expression ::= *expression* < *expression* | *expression* > *expression*
| *expression* <= *expression* | *expression* >= *expression*

equality_expression ::= *expression* == *expression* | *expression* != *expression*

logical_expression ::= ! *expression* | *expression* || *expression*
| *expression* && *expression*

bitwise_expression ::= ~ *expression* | ^ *expression*
| *expression* & *expression* | *expression* | *expression*
| *expression* << *expression* | *expression* >> *expression*

function_expression ::= *function_name*(*argument_list*$_{opt}$)
| (* *pointer*) (*argument_list*$_{opt}$)

argument_list ::= *expression* { , *expression* }$_{0+}$

type_name ::= *type_specifier* *declarator*$_{opt}$

B.6 Constant

constant ::= *character_constant* | *enumeration_constant* | *floating_constant*
| *integer_constant*

character_constant ::= ' *c* ' | L' *c* '

c ::= any character from the source character set except ' or \ or *newline*
| *escape_sequence*

escape_sequence ::= \' | \" | \? | \\ | \a | \b | \f | \n | \r | \t | \v
| \ *octal_digit* *octal_digit*$_{opt}$ *octal_digit*$_{opt}$
| \x *hexadecimal_digit* { *hexadecimal_digit* }$_{0+}$

enumeration_constant ::= *identifier*

string_literal ::= "{*character_constant*}$_{0+}$" | L *string_literal*

floating_constant ::= *fractional_constant* *exponential_part*$_{opt}$ *floating_suffix*$_{opt}$
digit_sequence *exponential_part* *floating_suffix*$_{opt}$

fractional_constant ::= *digit_sequence*$_{opt}$. *digit_sequence* | *digit_sequence* .

digit_sequence ::= { *digit* }$_{1+}$

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

exponential_part ::= { e | E }$_1$ { + | - }$_{opt}$ *digit_sequence*

floating_suffix ::= f | F | l | L

integer_constant ::= *decimal_constant* *integer_suffix*$_{opt}$
| *octal_constant* *integer_suffix*$_{opt}$
| *hexadecimal_constant* *integer_suffix*$_{opt}$

decimal_constant ::= 0 | *nonzero_digit* *digit_sequence*

nonzero_digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

octal_constant ::= 0 { *octal_digit* }$_{0+}$

octal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hexadecimal_constant ::= { 0x | 0X }$_1$ { *hexadecimal_digit* }$_{1+}$

hexadecimal_digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
a | b | c | d | e | f | A | B | C | D | E | F

integer_suffix ::= *unsigned_suffix* *long_suffix*$_{opt}$ | *long_suffix* *unsigned_suffix*$_{opt}$

unsigned_suffix ::= u | U

long_suffix ::= l | L

B.7 String Literal

string_literal ::= " *s_char_sequence* " | L" *s_char_sequence* "

s_char_sequence ::= { *sc* }$_{1+}$

sc ::= any character from the source character set except " or \ or *newline*
| *escape_sequence*

B.8 Preprocessor

preprocessing_directive ::= *control_line newline* | *if_section* | *pp_token newline*

control_line ::= `#` `include` { < *identifier* > | " *identifier* " }
| `#` `undef` *identifier* | `#` `line` *pp_token* | `#` `error` *pp_token*
| `#` `pragma` *pp_token*
| `#` `define` *identifier* { (*identifier_list*) }$_{opt}$ { *pp_token* }$_{0+}$

pp_token ::= *identifier* | *constant* | *string_literal* | *operator* | *punctuator*
| *pp_token* ## *pp_token* | # *identifier*

if_section ::= *if_group* { *elif_group* }$_{0+}$ { *else_group* }$_{opt}$ *end_if_line*

if_group ::= `#` `if` *const_integral_expr newline preprocessing_directive*$_{opt}$
| `#` `ifdef` *identifier newline* { *preprocessing_directive* }$_{opt}$
| `#` `ifndef` *identifier newline* { *preprocessing_directive* }$_{opt}$

elif_group ::= `#` `elif` *constant_expression newline* { *preprocessing_directive* }$_{opt}$

else_group ::= `#` `else` *newline* { *preprocessing_directive* }$_{opt}$

end_if_line ::= `#` `endif` *newline*

newline ::= the newline character

Appendix C

ANSI C Compared to Traditional C

In this appendix, we list the major differences between ANSI C and traditional C. Where appropriate, we have included examples. The list is not complete; only the major changes are noted.

C.1 Types

- The keyword `signed` has been added to the language.
- Three types of characters are specified: plain `char`, `signed char`, and `unsigned char`. An implementation may represent a plain `char` as either a `signed char` or an `unsigned char`.
- The keyword `signed` can be used in declarations of any of the signed integral types and in casts. Except with `char`, its use is always optional.
- In traditional C, the type `long float` is equivalent to `double`. Since `long float` was rarely used, it has been removed from ANSI C.
- The type `long double` has been added to ANSI C. Constants of this type are specified with the suffix `L`. A `long double` may provide more precision and range than a `double`, but it is not required to do so.
- The keyword `void` is used to indicate that a function takes no arguments or t returns no value.