Sintaxis de C

Ing. Jose Maria Sola – Dr. Oscar Ricardo Bruno

1.1. Gramática Léxica

1.1.1. Elementos Léxicos

```
<token> ->
     <palabra reservada> |
     <identificador> |
     <constante> |
     <literal de cadena> |
     <punctuator>
<token de preprocesamiento> ->
     <nombre de encabezado> |
     <identificador> |
     <número de preprocesador>|
     <constante carácter> |
     <literal de cadena> |
     <punctuator> |
     cada uno de los caracteres no-espacio-blanco que no sea uno de
     los anteriores
```

1.1.2. Palabras Reservadas

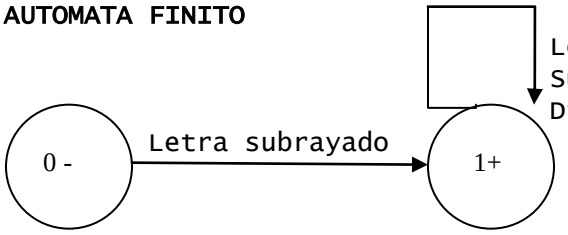
```
<palabra reservada> -> una de
     auto break case char const continue default do
     double else enum extern float for goto if
     int long register return short signed sizeof static
     struct switch typedef union unsigned void volatile while
```

1.1.3. Identificadores

```
<identificador> -> <no dígito> | <identificador> <no dígito> |
     <identificador> <dígito>
<no dígito> -> uno de _ a b c d e f g h i j k l m n o p q r s t u v
     w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
<dígito> -> uno de 0 1 2 3 4 5 6 7 8 9
```

- Dado que los identificadores constituyen un Lenguaje Regular, podemos describirlos mediante la definición regular.

```
<letra> = [a-zA-Z] (cualquier letra minúscula o mayúscula
     del alfabeto reducido)
<dígito> = [0-9]
<subrayado> = _
<primer carácter> = <letra> | <subrayado>
<otro carácter> = <letra> | <dígito> | <subrayado>
GRAMATICA
<identificador> = <primer carácter> <otro carácter>*
EXPRESION REGULAR
(letra + subrayado)(letra + subrayado + digito)
AUTOMATA FINITO
```



	E	L	_	D	O
0	1	1	--	--	
1	1	1	1	--	

```
int Columna (int);
int Automata (const char *cadena) {
static tablaT [3][4] = {{1,1,2,2},
                        {1,1,1,2},
                        {2,2,2,2}, /* rechazo */
                       };
   int estActual = 0;       /* estado inicial */
   unsigned int i = 0;         /* recorre la cadena */
   int caracter = cadena[0];  /* primer caracter */
   while (caracter != '\0' && estActual != 3) {
      estActual=tablaT[estActual][Columna(caracter)];
      caracter = cadena[++i];
   }
   if (estActual == 1) return 1; /* estado final */
   return 0;
}

int Columna (int c){
     if (c >= '0' && c <= '9') return 2;
     if (c == '_') return 1;
     if (isalpha(c) )return 0;
     return 3;
```

1.1.4. Constantes

```
<constante> ->
     <constante entera> |
     <constante real> |
     <constante carácter> |
     <constante enumeración>
```

- En general, en computación las constantes enteras *no* son un subconjunto de las constantes reales.

Constante Entera

```
<constante entera> ->
     <constante decimal> <sufijo entero>? |
     <constante octal> <sufijo entero>? |
     <constante hexadecimal> <sufijo entero>?
<constante decimal> ->
     <dígito no cero> |
     <constante decimal> <dígito>
<dígito no cero> -> uno de
     1 2 3 4 5 6 7 8 9
<dígito> -> uno de
     0 1 2 3 4 5 6 7 8 9
<constante octal> ->
     0 |
     <constante octal> <dígito octal>
<dígito octal> -> uno de
     0 1 2 3 4 5 6 7
<constante hexadecimal> ->
     0x <dígito hexadecimal> |
     0X <dígito hexadecimal> |
     <constante hexadecimal> <dígito hexadecimal>
<dígito hexadecimal> -> uno de
     0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
<sufijo entero> ->
     <sufijo "unsigned"> <sufijo "long">? |
     <sufijo "long"> <sufijo "unsigned">?
<sufijo "unsigned"> -> uno de
     u U
<sufijo "long"> -> uno de
     l L
```

- El tipo de una constante entera depende de su valor y será representada como primero corresponda, según la siguiente lista: **`int`**, **`unsigned int`**, **`long`**, **`unsigned long`**.

- El lenguaje de "Las constantes enteras en ANSI C" es regular; por lo tanto, podemos describirlo a través de una definición regular

```
<sufijo U> = u | U
<sufijo L> = l | L
<sufijo entero> =
     <sufijo U> |
     <sufijo L> |
     <sufijo U> <sufijo L> |
     <sufijo L> <sufijo U>
<dígito decimal> = [0-9]
<dígito decimal no nulo> = [1-9]
<dígito hexadecimal> = [0-9a-fA-F]
<dígito octal> = [0-7]
<prefijo hexadecimal> = 0x | 0X
<constante decimal> = <dígito decimal no nulo> <dígito
     decimal>*
<constante hexadecimal> = <prefijo hexadecimal> <dígito
     hexadecimal>+
<constante octal> = 0 <dígito octal>*
<constante incompleta> =
     <constante decimal> |
     <constante hexadecimal> |
     <constante octal>
<constante entera> = <constante incompleta> <sufijo
     entero>?
```

Constante Real

```
<constante real> ->
     <constante fracción> <parte exponente>? <sufijo real>?
     |
     <secuencia dígitos> <parte exponente> <sufijo real>?
<constante fracción> ->
     <secuencia dígitos>? . <secuencia dígitos> |
     <secuencia dígitos> .
<parte exponente> ->
     e <signo>? <secuencia dígitos> |
     E <signo>? <secuencia dígitos>
<signo> -> uno de + -
<secuencia dígitos> ->
     <dígito> |
     <secuencia dígitos> <dígito>
<dígito> -> uno de 0 1 2 3 4 5 6 7 8 9
<sufijo real> -> uno de f F l L
```

- Si no tiene sufijo, la constante real es **double**.

- En inglés esta constante es conocida como <floating-point-constant>, <constante de punto flotante>.

Constante Carácter

```
<constante carácter> ->
```

```
     '<carácter-c>' |
     '<secuencia de escape>'
<carácter-c> -> cualquiera excepto
     ' \
<secuencia de escape> ->
     <secuencia de escape simple> |
     <secuencia de escape octal> |
     <secuencia de escape hexadecimal>
<secuencia de escape simple> -> uno de
     \' \" \? \\ \a \b \f \n \r \t \v
<secuencia de escape octal> ->
     \<dígito octal> |
     \<dígito octal> <dígito octal> |
     \<dígito octal> <dígito octal> <dígito octal>
<dígito octal> -> uno de
     0 1 2 3 4 5 6 7
```

```
<secuencia de escape hexadecimal> ->
     \x<dígito hexadecimal> |
     \x <dígito hexadecimal> <dígito hexadecimal>
\x admite únicamente la x minúscula.
<dígito hexadecimal> -> uno de
     0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

Constante Enumeración

```
<constante enumeración> ->
     <identificador>
```

1.1.5. Constantes Cadena

```
<constante cadena> ->
     "<secuencia caracteres-s>"
<secuencia caracteres-s> ->
     <carácter-s> |
     <secuencia caracteres-s> <carácter-s>
<carácter-s> ->
     cualquiera excepto " \ |
     <secuencia de escape>
<secuencia de escape> ->
     <secuencia de escape simple> |
     <secuencia de escape octal> |
     <secuencia de escape hexadecimal>
<secuencia de escape simple> -> uno de
     \' \" \? \\ \a \b \f \n \r \t \v
<secuencia de escape octal> ->
     \ <dígito octal> |
     \ <dígito octal> <dígito octal> |
     \ <dígito octal> <dígito octal> <dígito octal>
<dígito octal> -> uno de
     0 1 2 3 4 5 6 7
<secuencia de escape hexadecimal> ->
     \x <dígito hexadecimal> |
     \x <dígito hexadecimal> <dígito hexadecimal
\x admite únicamente la x minúscula.
<dígito hexadecimal> -> uno de
     0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

- En inglés, estas constantes son referidas como <string literals>, <literales de cadena>.

- Notar que no están agrupadas con el resto de las constantes.

1.1.6. Punctuators – Caracteres de Puntuación

```
punctuator -> uno de
     [ ] ( ) { } . ->
     ++ -- & * + - ~ !
     / % << >> < > <= >= == != ^ | && ||
     = *= /= %= += -= <<= >>= &= ^= |=
```

? : ; ... , # ##

- La mayoría cumple el papel de operador, ver sección "*Precedencia y Asociatividad de los 45 Operadores*".

1.1.7. Nombre de Encabezados

```
<nombre de encabezado> ->
     < <secuencia de caracteres h> > |
     " <secuencia de caracteres q> "
<secuencia de caracteres h> ->
     <carácter h> |
     <secuencia de caracteres h> <carácter h>
<carácter h> ->
     cualquier miembro del conjunto de caracteres fuente excepto el
     carácter nueva-línea y el carácter >
<secuencia de caracteres q> ->
     <carácter q> |
     <secuencia de caracteres q> <carácter q>
<carácter q> ->
     cualquier miembro del conjunto de caracteres fuente excepto el
     carácter nueva-línea y el carácter "
```

1.1.8. Números de Preprocesador

```
<número de preprocesador> ->
     <dígito> |
     . <dígito> |
     <número de preprocesador> <dígito> |
     <número de preprocesador>  <identificador no dígito> |
     <número de preprocesador> e <sign> |
     <número de preprocesador> E <sign>|
     <número de preprocesador> p <sign>|
     <número de preprocesador> P <sign>|
     <número de preprocesador> .
```

1.2. Gramática de Estructura de Frases

1.2.1. Expresiones

```
<expresión> ->
     <expresión de asignación> |
     <expresión> , <expresión de asignación>
<expresión de asignación> ->
     <expresión condicional> |
     <expresión unaria> <operador asignación> <expresión de
     asignación>
<expresión condicional> ->
     <expresión O lógico> |
     <expresión O lógico> ? <expresión> : <expresión condicional>
<operador asignación> -> uno de
```

```
     = *= /= %= += -= <<= >>= &= ^= |=
<expresión O lógico> ->
     <expresión Y lógico> |
     <expresión O lógico> || <expresión Y lógico>
<expresión Y lógico> ->
     <expresión O inclusivo> |
     <expresión Y lógico> && <expresión O inclusivo>
<expresión O inclusivo> ->
     <expresión O excluyente> |
     <expresión O inclusivo> | <expresión O excluyente>
<expresión O excluyente> ->
     <expresión Y> |
     <expresión O excluyente> ^ <expresión Y>
<expresión Y> ->
     <expresión de igualdad> |
     <expresión Y> & <expresión de igualdad>
<expresión de igualdad> ->
     <expresión relacional> |
     <expresión de igualdad> == <expresión relacional> |
     <expresión de igualdad> != <expresión relacional>
<expresión relacional> ->
     <expresión de corrimiento> |
     <expresión relacional> < <expresión de corrimiento> |
     <expresión relacional> > <expresión de corrimiento> |
     <expresión relacional> <= <expresión de corrimiento> |
     <expresión relacional> >= <expresión de corrimiento>
<expresión de corrimiento> ->
     <expresión aditiva> |
     <expresión de corrimiento> << <expresión aditiva> |
     <expresión de corrimiento> >> <expresión aditiva>
<expresión aditiva> ->
     <expresión multiplicativa> |
     <expresión aditiva> + <expresión multiplicativa> |
     <expresión aditiva> - <expresión multiplicativa>
<expresión multiplicativa> ->
     <expresión de conversión> |
     <expresión multiplicativa> * <expresión de conversión> |
     <expresión multiplicativa> / <expresión de conversión> |
     <expresión multiplicativa> % <expresión de conversión>
<expresión de conversión> ->
     <expresión unaria> |
     (<nombre de tipo>) <expresión de conversión>
<expresión unaria> ->
     <expresión sufijo> |
     ++ <expresión unaria> |
     -- <expresión unaria> |
     <operador unario> <expresión de conversión> |
     sizeof <expresión unaria> |
     sizeof (<nombre de tipo>)
<nombre de tipo> está descripto más adelante, en la sección
     Declaraciones.
<operador unario> -> uno de & * + - ~ !
<expresión sufijo> ->
```

```
    <expresión primaria> |
    <expresión sufijo> [<expresión>] | /* arreglo */
    <expresión sufijo> (<lista de argumentos>?) | /* invocación */
    <expresión sufijo> . <identificador> |
    <expresión sufijo> -> <identificador> |
    <expresión sufijo> ++ |
    <expresión sufijo> --
<lista de argumentos> ->
    <expresión de asignación> |
    <lista de argumentos> , <expresión de asignación>
<expresión primaria> ->
    <identificador> |
    <constante> |
    <constante cadena> |
    (<expresión>)
```

Expresiones Constantes

```
<expresión constante> -> <expresión condicional>
```

- Las expresiones constantes pueden ser evaluadas durante la traducción en lugar de durante la ejecución.

1.2.2. Declaraciones

- Una declaración especifica la interpretación y los atributos de un conjunto de identificadores.

- Si una declaración provoca reserva de memoria, se la llama *definición*.

```
<declaración> ->
    <especificadores de declaración> <lista de declaradores>?
<especificadores de declaración> ->
    <especificador de clase de almacenamiento> <especificadores de
    declaración>? |
    <especificador de tipo> <especificadores de declaración>? |
    <calificador de tipo> <especificadores de declaración>?
<lista de declaradores> ->
    <declarador> |
    <lista de declaradores> , <declarador>
<declarador> ->
    <decla> |
    <decla> = <inicializador>
<inicializador> ->
    <expresión de asignación> | /* Inicialización de tipos
    escalares */
    {<lista de inicializadores>} | /* Inicialización de tipos
    estructurados */
    {<lista de inicializadores> , }
<lista de inicializadores> ->
    <inicializador> |
    <lista de inicializadores> , <inicializador>
```

```
<especificador de clase de almacenamiento> -> uno de
     typedef static auto register extern
```

- No más de un especificador de clase de almacenamiento> puede haber en una declaración

```
<especificador de tipo> -> uno de
     void char short int long float double signed unsigned
     <especificador de "struct" o "union">
     <especificador de "enum">
     <nombre de "typedef">
<calificador de tipo> -> const | volatile
<especificador de "struct" o "union"> ->
     <"struct" o "union"> <identificador>? {<lista de declaraciones
     "struct">} |
     <"struct" o "union"> <identificador>
<"struct" o "union"> -> struct | union
<lista de declaraciones "struct"> ->
     <declaración "struct"> |
     <lista de declaraciones "struct"> <declaración "struct">
<declaración "struct"> ->
     <lista de calificadores> <declaradores "struct"> ;
<lista de calificadores> ->
     <especificador de tipo> <lista de calificadores>? |
     <calificador de tipo> <lista de calificadores>?
<declaradores "struct"> ->
     <decla "struct"> |
     <declaradores "struct"> , <decla "struct">
<decla "struct"> ->
     <decla> |
     <decla>? : <expresión constante>
<decla> -> <puntero>? <declarador directo>
<puntero> ->
     * <lista calificadores tipos>? |
     * <lista calificadores tipos>? <puntero>
<lista calificadores tipos> ->
     <calificador de tipo> |
     <lista calificadores tipos> <calificador de tipo>
<declarador directo> ->
     <identificador> |
     ( <decla> ) |
     <declarador directo> [ <expresión constante>? ] |
     <declarador directo> ( <lista tipos parámetros> ) /* Declarador
     nuevo estilo */
     <declarador directo> ( <lista de identificadores>? ) /*
     Declarador estilo obsoleto */
<lista tipos parámetros> ->
     <lista de parámetros> |
     <lista de parámetros> , . . .
<lista de parámetros> ->
     <declaración de parámetro> |
     <lista de parámetros> , <declaración de parámetro>
<declaración de parámetro> ->
     <especificadores de declaración> <decla> | /* Parámetros
```

```
     "nombrados" */
     <especificadores de declaración> <declarador abstracto>? /*
     Parámetros "anónimos" */
<lista de identificadores> ->
     <identificador> |
     <lista de identificadores> , <identificador>
<especificador de "enum"> ->
     enum <identificador>? { <lista de enumeradores> } |
     enum <identificador>
<lista de enumeradores> ->
     <enumerador> | <lista de enumeradores> , <enumerador>
<enumerador> ->
     <constante de enumeración> |
     <constante de enumeración> = <expresión constante>
<constante de enumeración> -> <identificador>
<nombre de "typedef"> -> <identificador>
<nombre de tipo> -> <lista de calificadores> <declarador abstracto>?
<declarador abstracto> ->
     <puntero> |
     <puntero>? <declarador abstracto directo>
<declarador abstracto directo> ->
     ( <declarador abstracto> ) |
     <declarador abstracto directo>? [ <expresión constante>? ] |
     <declarador abstracto directo>? ( <lista tipos parámetros>? )
```

- Ejemplos de <nombre de tipo>:

```
int * [3] /* vector de 3 punteros a int */
int (*) [3] /* puntero a un vector de 3 ints */
int (*) ( void ) /* puntero a una función sin parámetros y
     que retorna un int */
int ( *[ ] ) ( unsigned, . . . ) /* vector de un número no
     especificado de punteros a funciones, cada una de las
     cuales tiene un  parámetro unsigned int más un número
     no especificado de otros parámetros, y retorna un int
     */
```

1.2.3. Sentencias

```
<sentencia> ->
     <sentencia expresión> |
     <sentencia compuesta> |
     <sentencia de selección> |
     <sentencia de iteración> |
     <sentencia etiquetada> |
     <sentencia de salto>
<sentencia expresión> ->
     <expresión>? ;
<sentencia compuesta> ->
     {<lista de declaraciones>? <lista de sentencias>?}
<lista de declaraciones> ->
     <declaración> |
```

```
     <lista de declaraciones> <declaración>
<lista de sentencias> ->
     <sentencia> |
     <lista de sentencias> <sentencia>
```

- La sentencia compuesta también se denomina *bloque*.

```
<sentencia de selección> ->
     if (<expresión>) <sentencia> |
     if (<expresión>) <sentencia> else <sentencia> |
     switch (<expresión>) <sentencia>
La expresión e controla un switch debe ser de tipo entero.
<sentencia de iteración> ->
     while (<expresión>) <sentencia> |
     do <sentencia> while (<expresión>) ;  |
     for (<expresión>? ; <expresión>? ; <expresión>?) <sentencia>
<sentencia etiquetada> ->
     case <expresión constante> : <sentencia> |
     default : <sentencia> |
     <identificador> : <sentencia>
Las sentencias case y default se utilizan solo dentro de una
     sentencia switch.
<sentencia de salto> ->
     continue ; |
     break ; |
     return <expresión>? ; |
     goto <identificador> ;
```

- La sentencia **continue** solo debe aparecer dentro del cuerpo de un ciclo. La sentencia **break** solo debe aparecer dentro de un **switch** o en el cuerpo de un ciclo. La sentencia **return** con una expresión no puede aparecer en una función **void**.

1.2.4. Definiciones Externas

```
<unidad de traducción> ->
     <declaración externa> |
     <unidad de traducción> <declaración externa>
<declaración externa> ->
     <definición de función> |
     <declaración>
```

- La unidad de texto de programa luego del preprocesamiento es una *unidad de traducción*, la cual consiste en una secuencia de declaraciones externas.

- Las *declaraciones externas* son llamadas así porque aparece fuera de cualquier función. Los términos *alcance de archivo* y *alcance externo* son sinónimos.

- Si la declaración de un identificador para un *objeto* tiene *alcance de archivo* y un *inicializador*, la declaración es una definición externa para el identificador.

```
<definición de función> ->
     <especificadores de declaración>? <decla> <lista de
     declaraciones>? <sentencia compuesta>
```

1.3. Gramática del Preprocesador

```
<archivo de preprocesamiento> ->
     <grupo>?
<grupo> ->
     <parte de grupo> |
     <grupo parte de grupo>
<parte de grupo> ->
     <sección if> |
     <línea de control> |
     <línea de texto> |
     # <no directiva>
<sección if> ->
     <grupo if> <grupos elif>? <grupo else>? <línea endif>
<grupo if> ->
     # if <expresión constante> <nueva línea> <grupo>? |
     # ifdef <identificador> <nueva línea> <grupo>? |
     # ifndef <identificador> <nueva línea>< grupo>?
<grupos elif> ->
     <grupo elif> |
     <grupos elif> <grupo elif>
<grupo elif> ->
     # elif <expresión constante> <nueva línea> <grupo>?
<grupo else> ->
     # else <nueva línea> grupo>?
<línea endif> ->
     # endif <nueva línea>
<línea de control> ->
     # include <tokens pp> <nueva línea> |
     # define <identificador> <lista de reemplazos> <nueva línea> |
     # define <identificador> <parizq> <lista de identificadores>? )
     <lista de reemplazos> <nueva línea> |
     # define <identificador> <parizq> ... ) <lista de reemplazos>
     <nueva línea> |
     # define <identificador> <parizq> <lista de identificadores> ,
     ... ) <lista de reemplazos> <nueva línea> |
     # undef <identificador> <nueva línea> |
     # line <tokens pp> <nueva línea> |
     # error <tokens pp>? <nueva línea> |
     # pragma <tokens pp> <nueva línea> |
     # <nueva línea>
<línea de texto> ->
     <tokens pp>? <nueva línea>
<no directiva> ->
     <tokens pp> <nueva línea>
<parizq> ->
     un carácter ( no inmediatamente precedido por un espacio blanco
<lista de reemplazos> ->
```

<tokens pp>?
<tokens pp> ->
<token de preprocesamiento> |
<tokens pp> <token de preprocesamiento>
<nueva línea> -> *el carácter nueva línea*
Las expresiones constantes de **if** y **elif** pueden estar formadas por los operadores comunes y/o por los siguientes operadores de preprocesamiento:

defined (<identificador>)
defined <identificador>
#
##