Practica 2.- Analizador Sintáctico de C-Dull

TALF 2018/19

11 de abril de 2019

Índice

1. Descripción de la práctica

Objetivo: El alumno deberá implementar un analizador sintáctico en Bison para un lenguaje basado en la sintaxis y las características más enervantes de los diferentes dialectos de C, al que llamaremos C-Dull. Usando el analizador léxico escrito para la Práctica 1, el programa resultante deberá recibir como argumento el path del fichero de entrada conteniendo el código fuente que se quiera analizar, y escribirá en la consola (o en un fichero) la lista de tokens encontrados en dicho fichero de entrada (omitiendo los comentarios) y las reglas reducidas. Estas últimas se volcarán a la salida a medida que se vayan reduciendo durante el análisis sintáctico.

Es necesario reducir todo lo que se pueda los conflictos en la gramática (o eliminarlos completamente, si es posible). En caso de que queden conflictos sin eliminar, se darán por buenos si la acción por defecto del analizador asegura un análisis correcto[1], y sois capaces de explicarme como funciona esa acción por defecto.

Documentación a presentar: El código fuente se enviará a través de Faitic. Para ello, primero se creará un directorio formado por los apellidos de los autores en orden alfabético, separados por un guión, y sin acentos ni eñes.

[1] Pista: dangling else.

Ej.- DarribaBilbao-DovalMosquera

Dentro del directorio se copiará el código fuente de la práctica: los archivos de Flex y Bison y cualquier otro archivo fuente (Makefile incluído) que se haya usado. A continuación, el directorio se comprimirá en un archivo (tar, tgz, rar o zip) con su mismo nombre.

Ej.- DarribaBilbao-DovalMosquera.zip

Grupos: Se podrá realizar individualmente o en grupos de dos personas.

Defensa: Consistirá en una demo al profesor, que calificará tanto los resultados como las respuestas a las preguntas que realice acerca de la implementación de la práctica.

Fecha de entrega y defensa: El código se subirá a Faitic como muy tarde el 2 de mayo de 2019 a las 10:00. La defensa tendrá lugar en las clases de aula pequeña los días 2, 6 y 8 de mayo.

Material: He dejado en Faitic (TALF → documentos y enlaces → Ejemplos prácticos (FLEX + BISON)) un fichero (`c-dull.tar.gz`). Dicho fichero contiene la especificación incompleta[2] en Flex del analizador léxico (`c-dull.l`), la especificación incompleta[3] del analizador sintáctico en Bison (`c-dull.y`, sólo con las reglas para definir un módulo), un fichero de ejemplo (`prueba.c`), y un Makefile.

Nota máxima: 2'5 ptos. Se evaluará al alumno por las partes del analizador que se hayan hecho satisfactoriamente:

- 0'4 por el espacio de nombres y variables.
- 0'4 por los tipos.
- 0'4 puntos por las clases y funciones.
- 0'5 puntos por las instrucciones.
- 0'5 por las expresiones.
- 0'3 por el tratamiento de errores.

Para sumar 0'5 por las expresiones, es necesario haberlas implementado correctamente usando una gramática no ambigua. Si se usan precedencias y asociatividades, la nota máxima por ese apartado se reduce a 0'25 ptos.

Si el analizador presentado tiene conflictos sin justificar, la nota de la práctica bajará 0'1 ptos por cada conflicto, hasta un máximo de 0'5.

2. Analizador léxico

Podéis utilizar el analizador definido en la práctica 1. Lo único que tenéis que hacer, a mayores, es añadir un return con el nombre del token correspondiente, en las acciones de las reglas, a continuación del printf con el que se sacaba el token encontrado por la consola. Lo único que voy a agregar son los nombres de los tokens (categorías léxicas) que están definidos en `c-dull.y`, y que tenéis que utilizar.

- Para los tokens formados por un único carácter, se usa ese carácter entre comillas simples.
- Para las palabras reservadas, el nombre de token será la palabra en mayúsculas. Por ejemplo, para '`float`', el nombre de token correspondiente será `FLOAT`, para '`goto`' será `GOTO`, etc. La única excepción son '`true`' y '`false`', que serán reconocidos como `BOOLEANO`.
- `CARACTER` corresponde a un carácter (incluyendo las comillas simples).
- `CADENA` corresponde a una cadena (incluyendo las comillas dobles). Fijaos que si habéis implementado la lectura de cadenas en flex con una condición de arranque y también la habéis llamado `CADENA`, el analizador resultante tendrá problemas. Podéis resolverlos cambiando el nombre a la condición de arranque.
- `IDENTIFICADOR` corresponde a un identificador.

[2] En realidad, prácticamente vacía.
[3] De nuevo, prácticamente vacía.

- `ENTERO` corresponde a un número entero.
- `REAL` corresponde a un número real.
- `BOOLEANO` corresponde a '`true`' y '`false`'.
- Los nombres para los operadores de más de un carácter son:

```
SUMA_ASIG '+='    RESTA_ASIG '-='   MULT_ASIG '*='   DIV_ASIG '/='  MOD_ASIG '%='
DESPI_ASIG '<<='  DESPD_ASIG '>>='  AND_ASIG '&='    OR_ASIG '|='   XOR_ASIG '^='
INC '++'          DEC '--'          DESPI '<<'       DESPD '>>'
GE '>='           LE '<='           EQ '=='          NEQ '!='
AND '&&'          OR '||'           PTR_ACCESO '->'
```

3. Gramática del analizador sintáctico de C-Dull

En esta sección vamos a proporcionar la especificación de la gramática de C-Dull. Para ello, usaremos una variante de la notación BNF, con la cabeza de cada regla separada de la parte derecha por el símbolo '`::=`'.

- Las palabras con caracteres en minúscula sin comillas denotan las categorías sintácticas (símbolos no terminales).
- Las secuencias de caracteres en mayúscula o entre comillas denotan las categorías léxicas (símbolos terminales). Por ejemplo, en:

```
expresion_parentesis ::= '(' expresion ')'
```

  `expresion` y `expresion_parentesis` son categorías sintácticas, mientras que '`(`' y '`)`' son categorías léxicas.
- Una barra vertical separa dos alternativas con el mismo símbolo cabeza. Por ejemplo:

```
expresion_constante ::= ENTERO | REAL | CADENA | CARACTER | BOOLEANO
```

  que también puede escribirse como:

```
expresion_constante ::= ENTERO
                      | REAL
                      | CADENA
                      | CARACTER
                      | BOOLEANO
```

- Los corchetes especifican la repetición de símbolos (terminales o no terminales)[4] . Dichos símbolos pueden aparecer un número finito de veces, como mínimo cero, en cuyo caso escribiremos '`[ ]*`', o una, en cuyo caso escribimos '`[ ]+`'. De este modo, en la siguiente regla:

```
modulo ::= [ directiva_uso ]* [ declaracion ]+
```

  tenemos que un módulo está formado por 0 o más directivas de uso y 1 o más declaraciones.
- Los paréntesis especifican la repetición de símbolos separados por comas[5]. Dichos símbolos pueden aparecer un número finito de veces, como mínimo cero, en cuyo caso escribiremos '`( )*`', o una, en cuyo caso escribimos '`( )+`'. De este modo, la siguiente regla:

```
valor ::= '{' ( valor )+ '}'
```

[4]Excepto cuando aparecen entre comillas. En ese caso son categorías léxicas.
[5]Excepto cuando aparecen entre comillas.

especifica que un `valor` deriva 1 o más símbolos `valor`, separados por comas, entre llaves (los símbolos terminales '`{`' y '`}`').

- Los símbolos '`[ ]?`' delimitan los elementos opcionales. En el siguiente ejemplo:

```
declaracion_miembro_enum ::= IDENTIFICADOR [ '=' expresion ]?
```

el terminal '`=`' y el no terminal `expresion` son opcionales, por lo que puede aparecer o no en una declaración (aunque si lo hacen tienen que aparecer ambos).

3.1. Bloques constituyentes de un módulo C-Dull (ya hechos en `c-dull.y`)

Un modulo C-Dull está compuesto por 0 o más directivas de uso seguidas de 1 o más declaraciones. A su vez las declaraciones pueden ser de espacios de nombre, variables, tipos o funciones.

```
modulo ::= [ directiva_uso ]* [ declaracion ]+

declaracion ::= declaracion_espacio_nombres
              | declaracion_variables
              | declaracion_tipo
              | declaracion_funcion

directiva_uso ::= 'using' [ IDENTIFICADOR '=' ]? nombre_tipo_o_espacio_nombres ';'

nombre_tipo_o_espacio_nombres ::= [ identificador_con_tipos '.' ]* identificador_con_tipos

identificador_con_tipos ::= IDENTIFICADOR [ '(' ( nombre_tipo_o_espacio_nombres )+ ')' ]?
```

Por su parte, una directiva de uso especifica los módulos externos cuyos contenidos (tipos de datos, funciones, objetos, etc) van a ser invocados en el módulo actual. Por lo tanto está formado por la palabra reservada '`using`' seguida de un nombre de tipo o espacio de nombres. Previamente a este último, puede aparecer de forma opcional un identificador seguido del operador '`=`', que se usará como el nombre local del módulo que se carga.

Un nombre de tipo o espacio de nombres estará formado por uno más identificadores con tipos, separados por puntos, y un identificador con tipo está formado por un identificador y, opcionalmente uno o más nombres de tipo o espacio de nombres, entre paréntesis y separados por comas.

3.2. Espacios de nombres

Un espacio de nombres se utiliza para declarar un ámbito que define un conjunto de tipos de datos, variables, objetos, etc, relacionados entre si. Se declara con la palabra reservada '`namespace`' seguida de un identificador (el nombre del ámbito declarado) y un bloque de espacio de nombres.

```
declaracion_espacio_nombres ::= 'namespace' identificador_anidado bloque_espacio_nombres

identificador_anidado ::= [ IDENTIFICADOR '.' ]* IDENTIFICADOR

bloque_espacio_nombres ::= '{' [ directiva_uso ]* [ declaracion ]+ '}'
```

A su vez, un identificador anidado está formado por uno o más identificadores separados por puntos, mientras que los bloques de espacios de nombres están formados por 0 o más directivas de uso y 1 o más declaraciones (de espacios de nombres anidados, tipos, variables o funciones).

3.3. Declaraciones de variables

Una declaración de variable está formada por un tipo (nombre de tipo o tipo predefinido) seguido de uno o más nombres de variables separadas por comas, terminando en un punto y coma. Por su parte, un tipo puede ser un nombre de tipo o espacio de nombres entre '<' y '>' (el nombre de un tipo declarado previamente), o un tipo escalar.

```
declaracion_variable ::= tipo ( nombre )+ ';'

tipo ::= '<' nombre_tipo_o_espacio_nombres '>'
       | tipo_escalar
```

Esto último supone una diferencia entre C-Dull y los dialectos de C. Los analizadores léxicos de los compiladores de C, cada vez que encuentran un identificador, suelen comprobar si se trata de un tipo de dato, variable o macro, y devolver un token distinto en cada caso. Para esa comprobación hacen uso de una estructura de datos que no hemos visto y, por lo tanto, no usaremos: la tabla de símbolos. Eso quiere decir que nuestro analizador léxico no puede distinguir entre un identificador que es un nombre de variable y un identificador que es un nombre de tipo. Ello da lugar a conflictos en el analizador cuando se intenta reconocer definiciones de variables, y para prevenirlos, cuando en C-Dull se define una variable usando un tipo definido previamente, el nombre del mismo se pone entre '<' '>'. Por ejemplo:

```
typedef natural unsigned int;
<natural> n;
```

En este fragmento de código estaríamos definiendo el tipo `natural` como un entero sin signo y, a continuación, declarando una variable `n` de tipo `natural`. En general, cuando queramos utilizar el nombre de un tipo definido previamente por el usuario (por ejemplo en la definición de una función/método), tendremos que ponerlo entre '<' '>'. Las únicas excepciones a esta regla son la definición de herencia en las declaraciones de interfaz o clase.

Por otra parte, un tipo escalar está compuesto por un tipo básico ('`char`', '`int`', '`float`', '`double`' y '`boolean`') precedido, opcionalmente, de un especificador de longitud ('`short`' o '`long`') y/o un especificador de signo ('`signed`' o '`unsigned`'). Aunque hay especificadores de longitud y signo que no tiene sentido aplicar a ciertos tipos (por ejemplo, '`long char`' o '`unsigned boolean`'), hemos elegido no incorporar esas incompatibilidades a la gramática para no hacerla demasiado compleja (es decir, tenemos una gramática que sobregenera). Supondremos que la comprobación de dichas incompatibilidades se haría en el análisis semántico, que no vamos a implementar.

```
tipo_escalar ::= [ signo ]? [ longitud ]? tipo_basico

longitud ::= SHORT | LONG

signo ::= SIGNED | UNSIGNED

tipo_basico ::= CHAR | INT | FLOAT | DOUBLE | BOOLEAN

nombre ::= dato [ '=' valor ]?

dato ::= [ '*' ]* dato_indexado

dato_indexado ::= IDENTIFICADOR [ '[' ( expresion )* ']' ]*

valor ::= expresion
        | '{' ( valor )+ '}'
```

Un `nombre` está formado por un `dato` seguido, opcionalmente, de un operador '=' seguido de un `valor`. A su vez un `dato` está compuesto de 0 o más '*' (operadores de punteros) y un `dato_indexado`, que estará formado por un identificador seguido de uno o más corchetes '[' ']' encerrando una o más expresiones separadas por comas. La idea es que si hay varios corchetes el tipo es un array de arrays, mientras que si tenemos varias expresiones dentro de un par de corchetes, tenemos un array multidimensional.

```
int lista1[10,5];  // array de 2 dimensiones
int lista2[10][];  // array de arrays
```

La diferencia entre ambos tipos de array es que en el primer caso tenemos que especificar las dimensiones del array, de modo que todas las líneas y columnas de `lista1` tendrán las misma longitud, mientras que en `lista2` sólo tenemos que declarar la dimensión del array principal, por lo que que los arrays contenidos en éste pueden ser de distintas longitudes.

Finalmente, un valor estará formado por una `expresion` o una lista de valores separados por comas entre llaves. Fijaos que cada uno de estos valores puede ser, a su vez, una lista de valores entre llaves.

```
int lista[2][] = {{1,2,3}, {4,5}};
```

3.4. Tipos

En esta sección vamos a ver las formas de declarar nuevos tipos, al margen de los predefinidos por el lenguaje y de las clases, que tendrán su propia descripción aparte. En principio, las declaraciones de tipo pueden derivar declaraciones de nombramientos de tipo, structs y unions, interfaces, tipos enumerados y clases.

```
declaracion_tipo ::= nombramiento_tipo
                   | declaracion_struct_union
                   | declaracion_interfaz
                   | declaracion_enum
                   | declaracion_clase
```

A su vez, un nombramiento de tipo (consistente en dar un nombre a un tipo ya existente) está formado por la palabra reservada '`typedef`', seguida del tipo que vamos a nombrar y un identificador (el nuevo nombre del tipo). Finalizamos la declaración con un punto y coma.

```
nombramiento_tipo ::= 'typedef' tipo ID ';'
```

Por otro lado, podemos definir el tipo de dato struct o union de la misma manera, escribiendo la palabra reservada respectiva, un identificador opcional como el nombre del tipo de dato, y listando los campos que componen el tipo entre llaves (a través del no terminal `declaracion_struct`). Opcionalmente, podemos especificar 0 o más modificadores al principio de la declaración ('`new`', '`public`', '`protected`', '`internal`', '`private`', '`static`', '`virtual`', '`sealed`', '`override`', '`abstract`', '`extern`') [6].

```
declaracion_struct_union ::= [ modificador ]* struct_union [ IDENTIFICADOR ]?
                             '{' [ declaracion_campo ]+ '}'

modificador ::= 'new' | 'public' | 'protected' | 'internal' | 'private' | 'static'
              | 'virtual' | 'sealed' | 'override' | 'abstract' | 'extern'

struct_union ::= 'struct' | 'union'

declaracion_campo ::= tipo ( nombre )+ ';'
                    | declaracion_struct_union ( nombre )+ ';'
```

A su vez, el no terminal `declaracion_struct` permite declarar uno o más campos especificando su tipo seguido de una lista de nombres separados por comas, con la misma categoría sintáctica `nombre` definida en el apartado anterior. El tipo puede ser un tipo escalar del lenguaje, un array, un tipo definido por el usuario o una struct o union. Por ejemplo:

```
struct miStruct {
  int campoArray[];
  union {
```

[6] En realidad, sólo los modificadores '`new`', '`public`', '`protected`', '`internal`' y '`private`' son aplicables a las struct/union, pero tener modificadores distintos para structs, interfaces, tipos enumerados y clases haría que la gramática no fuera LALR(1), por lo que supondremos que la determinación de si los modificadores son correctos se realizaría durante el análisis semántico.

```
    char *subcampoUno = "hola";
    <predefinido> subcampoDos = -134;
    } campoUnion;
  }
```

Respecto a las declaraciones de interfaz, se utilizan para definir especificaciones de métodos que deberán ser cumplidas por las clases que implementen dicho interfaz. Están compuestas por cero o varios modificadores[7], seguidos por la palabra clave '`interface`', el nombre del interfaz (un identificador), el delimitador '`:`', y la lista de interfaces de las que hereda propiedades.

```
  declaracion_interfaz ::= [ modificador ]* 'interface' IDENTIFICADOR herencia cuerpo_interfaz

  herencia ::= [ ':' ( nombre_tipo_o_espacio_nombres )+ ]?

  cuerpo_interfaz ::= '{' [ declaracion_metodo_interfaz ]* '}'

  declaracion_metodo_interfaz ::= [ 'new' ]? firma_funcion ';'
```

Finalmente, se escribe entre llaves el cuerpo de la interfaz, consistente en 0 o más declaraciones de métodos de interfaz. Estos últimos están compuestos de una firma de función (que veremos en la sección dedicada a funciones), opcionalmente precedida por el modificador '`new`'. Por ejemplo:

```
public interface arbolBinario : arbol {
    boolean vacio();
    <arbolBinario> hijoDcho();
    <arbolBinario> hijoIzdo();
    boolean esta(<elem> elemento);
    void setRaiz(<elem> elemento);
    void setHijoDcho(<arbolBinario> arbol);
    void setHijoIzdo(<arbolBinario> tree);
    void vaciar();
    }
```

Los tipos enumerados están, como su nombre indica, definidos como una lista de posibles miembros del tipo. Por lo tanto, la declaración de un tipo enumerado está compuesta de 0 o más modificadores [8], la palabra reservada '`enum`', el nombre del tipo (un identificador) y el cuerpo de la declaración. Opcionalmente, entre estos dos ultimos elementos, se puede declarar el tipo predefinido al que pertenecen los elementos del tipo enumerado, usando el delimitador '`:`' seguido del nombre del tipo.

```
  declaracion_enum ::= [ modificador ]* 'enum' IDENTIFICADOR [ ':' tipo_escalar ]? cuerpo_enum

  cuerpo_enum ::= '{' ( declaracion_miembro_enum )+ '}'

  declaracion_miembro_enum ::= IDENTIFICADOR [ '=' expresion ]?
```

El cuerpo de la declaración de tipo enumerado está formado por la definición de uno o más miembros del tipo enumerado, entre llaves. Cada miembro tendrá un nombre (un identificador), seguido opcionalmente del operador '`=`' y el valor del miembro del tipo enum, formado por una expresión del mismo tipo que el `tipo_escalar` declarado tras el nombre de tipo enumerado [9]. Por ejemplo:

```
enum color: int {
  Rojo = 1, Verde = 2, Azul = 3
  }
```

[7] Al igual que en el caso de las structs, sólo los modificadores '`new`', '`public`', '`protected`', '`internal`' y '`private`' son aplicables.
[8] Concretamente, '`new`', '`public`', '`protected`', '`internal`' y '`private`'.
[9] Esa comprobación de tipos se haría en el análisis semántico, así que podéis ignorarla.

3.5. Clases

Una declaración de clase está formada por una lista de 0 o más modificadores[10] seguidos de la palabra reservada '`class`', un identificador como nombre de la clase, la lista de clases o interfaces de las que la clase definida hereda y el cuerpo de la clase. A su vez, el cuerpo de la clase está formado por una o más declaraciones de atributos, métodos, tipos, constructores o destructor, delimitados por '`{`' '`}`'.

```
declaracion_clase ::= [ modificador ]* 'class' IDENTIFICADOR herencia cuerpo_clase

cuerpo clase ::= '{' [ declaracion_elemento_clase ]+ '}'

declaracion_elemento_clase ::= declaracion_tipo
                             | declaracion_atributo
                             | declaracion_metodo
                             | declaracion_constructor
                             | declaracion_destructor
                             | declaracion_atributo
```

Ya hemos descrito la sintaxis de una declaración de tipo previamente. Por su parte, una declaración de atributo está formado por 0 o más modificadores [11], seguida de una declaración de variable. Una declaración de método está formada por 0 o más modificadores [12] seguidos de una firma de función y un bloque de instrucciones (ambos serán explicados más adelante).

```
declaracion_atributo ::= [ modificador ]* declaracion_variable

declaracion_metodo ::= [ modificador ]* firma_funcion bloque_instrucciones

declaracion_constructor ::= [ modificador ]* cabecera_constructor bloque_instrucciones

cabecera_constructor ::= IDENTIFICADOR parametros [ inicializador_constructor ]?

inicializador_constructor ::= ':' BASE parametros
                            | ':' THIS parametros

declaracion_destructor ::= [ modificador ]* cabecera_destructor bloque_instrucciones

cabecera_destructor ::= '~' IDENTIFICADOR '(' ')'
```

Por su parte, un método constructor se declara mediante 0 o más modificadores [13] seguidos de la cabecera del constructor y un bloque de instrucciones. A su vez, la cabeza del constructor está formada por un identificador (el nombre del constructor [14]), los parámetros del método (cuya definición se verá en la siguiente sección), y, opcionalmente, la llamada a otro constructor que se ejecutará al comienzo de la ejecución del que se está definiendo. Si dicho constructor se designa con la palabra reservada '`base`' seguida de una serie de parámetros, se estará referenciando a un constructor de la clase de la que hereda directamente la actual, mientras que si se define con '`this`', se estará haciendo referencia a otro constructor de la clase actual.

Finalmente, el método destructor de una clase se especifica como cero o más modificadores [15], seguidos de la cabecera del destructor y un bloque de instrucciones. La cabecera estará formada por '`~`' seguido del nombre del destructor (un identificador) y los paréntesis '`(` `)`', dado que se trata de un método sin parámetros.

Un pequeño ejemplo:

[10] En este caso, '`new`', '`public`', '`protected`', '`internal`', '`private`', '`abstract`', '`sealed`' o '`static`'.
[11] En este caso, `new`, `public`, `protected`, `internal`, `private`, `static`, `readonly` y `volatile`.
[12] En este caso, `new`, `public`, `protected`, `internal`, `private`, `static`, `virtual`, `sealed`, `override`, `abstract` y `extern`.
[13] En este caso, `public`, `protected`, `internal`, `private` y `extern`.
[14] Que tiene que ser igual al nombre de la clase, lo que tendría que chequear el analizador semántico.
[15] En este caso, sólo `extern`. Aunque sería más sencillo poner `[ 'extern' ]?`, eso daría lugar a conflictos en la gramática. De nuevo, suponemos que el análisis semántico se encargará de admitir sólo el modificador permitido.

```
public class esfera : forma {
   private float radio;
   esfera (float radio) : base () {
     esfera.radio = radio;
     }
   public float volumen() {
     return (4/3*PI*pow(radio,3));
     }
   ~destructor () {}
   }
```

3.6. Funciones

Una declaración de función está formada por una firma seguida de un bloque de instrucciones. A su vez, una firma se especifica como un tipo (que puede ser la palabra reservada '`void`' para denotar que no hay valor de retorno, o una secuencia derivada de `tipo` seguido de 0 o más operadores de puntero '`*`'), seguida de un identificador (el nombre de la función) y los `parametros` de la misma. Dichos `parametros`, se definen como 0 o más `argumentos` entre paréntesis separados por '`;`'.

```
declaracion_funcion ::= firma_funcion bloque_instrucciones

firma_funcion ::= VOID IDENTIFICADOR parametros
                | tipo [ '*' ]* IDENTIFICADOR parametros

parametros ::= '(' [ argumentos ';' ]* [ argumentos ]? ')'

argumentos ::= nombre_tipo ( variable )+

nombre_tipo ::= tipo [ '*' ]*

variable ::= IDENTIFICADOR [ '=' expresion ]?
```

Finalmente, cada definición de `argumentos` consta de un `nombre_tipo` (un `tipo` con 0 o más '`*`'), seguido por 1 o más `variables`. Estos son identificadores separados por comas, que, opcionalmente, pueden tener asociado un valor por defecto, a través del símbolo '`=`' y una expresión. Por ejemplo:

```
float areaRectangulo(float Base=1,altura=1) { // ponemos Base en mays
  return Base*altura;                         // dado que base es palabra
  }                                           // reservada

void altaUsuario(char *nombre,direccion; int edad; float peso,altura) {
  // algo de codigo aqui
  }
```

3.7. Instrucciones

Vamos a considerar los siguientes tipos de instrucciones:

```
instruccion ::= bloque_instrucciones
              | instruccion_expresion
              | instruccion_bifurcacion
              | instruccion_bucle
              | instruccion_salto
              | instruccion_destino_salto
              | instruccion_retorno
```

```
              | instruccion_lanzamiento_excepcion
              | instruccion_captura_excepcion
              | instruccion_vacia
```

Un bloque de instrucciones, delimitado por los símbolos '`{`' y '`}`', está formado por por 0 o más declaraciones seguidas de 0 o más instrucciones. Por lo tanto, podemos tener bloques de instrucciones vacíos.

```
bloque_instrucciones ::= '{' [ declaracion ]* [ instruccion ]* '}'
```

Una `instruccion_expresion` está formada bien por una llamada a función (cuya definición veremos en la siguiente sección) o bien por una asignación. Dicha asignación está, a su vez, compuesta por una expresión indexada (que veremos en la siguiente sección) seguida de un operador de asignación y una expresión.

```
instruccion_expresion ::= expresion_funcional ';' | asignacion ';'

asignacion ::= expresion_indexada operador_asignacion expresion

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

Las instrucciones de bifurcación son de dos tipos: `if-else` y `switch-case-default`. La primera de ellas está definida como en C, con la palabra reservada '`if`', seguida de una expresion entre paréntesis (la condición del `if`), y una instrucción. Opcionalmente, podemos tener un '`else`' seguido de una instrucción. Recordad que `instruccion` puede derivar una instrucción o un bloque de instrucciones.

```
instruccion_bifurcacion ::= 'if' '(' expresion ')' instruccion [ 'else' instruccion ]?
                          | 'switch' '(' expresion ')' '{' [ instruccion_caso ]+ '}'

instruccion_caso ::= 'case' expresion ':' instruccion
                   | 'default' ':' instruccion
```

En el caso de tener una situación similar a la del siguiente ejemplo (un `if` con un `if-else` anidado como instrucción a ejecutar si se cumple la condición), el `else` pertenecerá al `if` más interno.

```
if (i>0)
  if (i<9)
    i=i+1;
  else      // este else corresponde al if mas interno
    i= 10;
```

Por su parte, las instrucciones `switch-case-default` están compuestas por la palabra reservada '`switch`', seguida de una expresión entre paréntesis. A continuación, entre llaves, tendremos cada uno de los posibles casos de la expresión condicional. Dichos casos están compuestos por la palabra reservada '`case`', seguida de una expresión, el delimitador '`:`' y una instrucción.

Respecto a los bucles, tenemos tres tipos: `while`, `do-while` y `for`, que se resumen a continuación.

```
instruccion_bucle ::= 'while' '(' expresion ')' instruccion
                    | 'do' instruccion 'while' '( expresion ')' ';'
                    | 'for' '(' ( asignacion )* ';' expresion ';' ( expresion )+ ')' instruccion
```

Un bucle `while` está formado por la palabra reservada, seguida de una expresión entre paréntesis (la condición del bucle) y una instrucción. La sintaxis del bucle `do-while` se define como la palabra reservada '`do`', seguida de una instrucción, la palabra reservada '`while`', la condición del bucle especificada igual que en el bucle `while` y un delimitador '`;`'. En el bucle `for` tenemos la palabra reservada '`for`', y, entre paréntesis, 0 o más inicializaciones de las variables índice de bucle, la condición de parada, y 1 o más incrementos/decrementos de las variables índice, los tres separados por '`;`'. Tanto las inicializaciones de las variables índice del bucle como sus incrementos estarán separados por comas.

También tenemos una instrucción general de salto incondicional, con la palabra reservada '`goto`', seguida del nombre de la etiqueta (un identificador) que designa la instrucción destino del salto. Por lo tanto, también necesitamos alguna manera de especificar el destino del salto asociando un identificador a una instrucción.

```
instruccion_salto ::= 'goto' IDENTIFICADOR ';' | 'continue' ';' | 'break' ';'

instruccion_destino_salto ::= IDENTIFICADOR ':' instruccion ';'
```

Además, tenemos dos intrucciones de salto cuya única finalidad es salir de la iteración actual de un blucle: '`continue`', para ir a la iteración siguiente, y ('`break`') para saltar a la instrucción siguiente al bucle.

También debemos especificar la intrucción de retorno de una función, formada por la palabra reservada '`return`', seguida, opcionalmente, de una expresión, correspondiente al valor de retorno, y el delimitador '`;`'.

```
instruccion_retorno ::= 'return' [ expresion ]? ';'
```

También se hacen necesarias dos instrucciones para el lanzamiento y captura de excepciones. Para el lanzamiento, tenemos la palabra reservada '`throw`' seguida de una expresión y '`;`'. Para la captura de excepciones, tenemos la instrucción `try-catch-finally`. La sintaxis de esta última instrucción está formada por la palabra reservada '`try`' seguida de un bloque de instrucciones, y de 1 o más clausulas catch o una clausula finally (o ambas). A su vez, una clausula catch puede ser específica o general. En el primer caso, acompañanando a la palabra reservada '`catch`', tendremos un tipo excepción entre paréntesis (que se omite en el caso de una cláusula catch general) seguido de un bloque de instrucciones.

```
instruccion_lanzamiento_excepcion ::= 'throw' expresion ';'

instruccion_captura_excepcion ::= 'try' bloque_instrucciones clausulas-catch
                                | 'try' bloque_instrucciones clausula_finally
                                | 'try' bloque_instrucciones clausulas-catch clausula_finally

clausulas-catch ::= [ clausula_catch_especifica ]+
                  | clausula_catch_general
                  | [ clausula_catch_especifica ]+ clausula_catch_general

clausula_catch_especifica ::= 'catch' '(' nombre_tipo ')' bloque_instrucciones

clausula_catch_general ::= 'catch' bloque_instrucciones

clausula-finally ::= 'finally' bloque_instrucciones
```

Es posible tener varias clausulas específicas, una sola cláusula general o ambas. En este último caso, la clásula general debe aparecer después de las específicas. Por último, la cláusula '`finally`' está compuesta de la palabra reservada seguida de un bloque de instrucciones.

Finalmente, la instrucción vacía está formada por el delimitador de instrucción '`;`'.

```
instruccion_vacia ::= ';'
```

3.8. Expresiones

Para definir las posibles expresiones matemáticas y lógicas (que tendrán como raíz el símbolo no terminal `expresion`), primero vamos a especificar los posibles operandos. Los más simples son las constantes y expresiones entre paréntesis.

```
expresion_constante ::= ENTERO | REAL | CADENA | CARACTER | BOOLEANO

expresion_parentesis ::= '(' expresion ')'
```

Aquí tenemos un compromiso entre simplicidad de la gramática y sobregeneración. Al tener cadenas como posibles valores constantes en una expresión, podríamos aplicarles operadores que no les corresponden, como, por ejemplo, operadores aritméticos. De nuevo, podemos reescribir la gramática, o retrasar la solución al problema hasta el chequeo

de tipos, lo que parece razonable, dado que tenemos que hacer dicho chequeo de todas maneras. En vuestro caso, tenéis que implementar la especificación tal como está.

En otro orden de cosas, los operandos también pueden ser elementos de arrays, campos o atributos de tipos de datos complejos (o de punteros a tipos de datos complejos), valores de retorno de funciones, creación de nuevos objetos, etc

```
expresion_funcional ::= identificador_anidado '(' ( expresion )* ')'

expresion_creacion_objeto ::= 'new' identificador_anidado '(' ( expresion )* ')'

expresion_indexada ::= identificador_anidado
                     | expresion_indexada '[' expresion ']'
                     | expresion_indexada '->' identificador_anidado
```

A continuación tenemos los operadores unarios. Para evitar conflictos, los hemos separado en postfijos y prefijos.

```
expresion_postfija ::= expresion_constante
                     | expresion_parentesis
                     | expresion_funcional
                     | expresion_creacion_objeto
                     | expresion_indexada
                     | expresion_postfija '++'
                     | expresion_postfija '--'

expresion_prefija ::= expresion_postfija
                    | 'sizeof' expresion_prefija
                    | 'sizeof' '(' nombre_tipo ')'
                    | operador_prefijo expresion_cast

operador_prefijo ::= '++' | '--' | '&' | '*' | '+' | '-' | '~' | '!'
```

También implementaremos la posibilidad de hacer casts sobre operandos. Para ello, debemos definir un tipo o nombre de tipo entre paréntesis.

```
expresion_cast ::= expresion_prefija
                 | '(' nombre_tipo ')' expresion_prefija
```

A partir de aquí, tenéis que implementar el resto de operadores binarios de manera similar a como lo hemos hecho, teniendo en cuenta sus precedencias y asociatividades. De mayor a menor precedencia:

- '*', '/' y '%'
- '+' y '-'
- '<<' y '>>' (operadores de desplazamiento)
- '&' (and binario)
- '^' (xor binario)
- '|' (or binario)
- '<', '>', '<=', '>='
- '==' y '!='
- '&&' (and lógico)
- '||' (or lógico)

La asociatividad de todos estos operadores es por la izquierda.

A la hora de diseñar las reglas para los operadores binarios, podéis implementar la precedencia y asociatividad diseñando una gramática determinista, o podéis implementar esta porción de la gramática como ambigua y definir las precedencias y asociatividades a través de la definición de los operadores en la zona de declaraciones. No he probado a hacerlo de la segunda forma, por lo que no sé si será más complicado, o si el hecho de que haya operadores sobrecargados (por ejemplo, '`*`' puede ser el producto o un puntero) os va a plantear problemas. Mi consejo es que implementéis esta porción de la especificación como una gramática determinista. Si lo seguís, podéis usar como ejemplo la gramática de las expresiones aritméticas que se usa repetidamente en los ejemplos sobre gramáticas LR(k) que hemos visto en clase:

$$\begin{array}{llll} E' \to E & E \to E + T & T \to T * F & F \to (E) \\ & \mid T & \mid F & \mid id \end{array}$$

También debéis recordar que la asociatividad de un operando determina el tipo de recursividad que vais a usar en las reglas para implementar dicho operando. Así, si es asociativo por la izquierda, la regla o reglas correspondientes serán recursivas por la izquierda, como ocurre con la suma y el producto en la gramática anterior. Por otra parte, si el operador es asociativo por la derecha, se implementará mediante reglas con recursividad derecha. Finalmente, si el operador no es asociativo se implementará a través de reglas no recursivas.

Una vez implementados los operadores anteriores, si llamamos al simbolo raíz de la porción de la gramática correspondiente a todos los operadores y operandos anteriores `expresion_logica`, podemos terminar de definir una `expresion` implementando el operador condicional ternario:

```
expresion ::= expresion_logica [ '?' expresion ':' expresion ]?
```

3.9. Implementación

En las secciones anteriores he intentado presentar la especificación de la gramática de la manera más clara posible. En general he presentado las reglas de arriba (más cerca del axioma) a abajo, excepto en el caso de las expresiones, en el que debido a la complejidad en el número de operadores potenciales, y a la necesidad de implementar la precedencia y asociatividad de los mismos, me pareció mejor idea empezar por la parte más baja, la definición de operandos, antes de pasar a los operadores.

Ahora bien, que haya definido la especificación de C-Dull en un cierto orden, no quiere decir que sea el mejor orden para escribir las reglas del analizador. En vuestro lugar, yo intentaría escribir la gramática por partes, y hacer pruebas sobre lo ya escrito antes de pasar a la siguiente parte. El orden que yo seguiría es:

1. Expresiones.
2. Instrucciones (empezando por la `instruccion_expresion`).
3. Especificación de funciones.
4. Declaraciones de tipos de datos, variables y espacios de nombres.

Este procedimiento tiene la ventaja de que, si no os da tiempo a implementar toda la gramática, podéis presentarme una porción de la misma que funcione. Además, en caso de hacer la práctica por parejas, el escribir las reglas de esta manera os permite trabajar en paralelo en diferentes partes de la especificación. Por ejemplo, podéis hacer las expresiones y las instrucciones al mismo tiempo. El encargado de escribir las instrucciones puede tener una regla para que la categoría `expresion` derive, por ejemplo, un `REAL`, mientras su compañero no escriba las reglas correspondientes.

3.10. Depuración de la gramática

Dado que, como se establece en la siguiente sección, tenéis que volcar en la consola (o fichero) las reglas que se van reduciendo, podéis usar esa información para depurar la gramática. Si ello no es suficiente, os recomiendo que uséis la macro `YYDEBUG`, para lo que tenéis que seguir dos pasos:

1. Declarar dicha macro en la sección de declaraciones de Bison (lo que ya está hecho en el esqueleto de analizador sintáctico que os hemos pasado en `c-dull.tar.gz`).

```
%{

  #include <stdio.h>
  extern FILE *yyin;
  extern int yylex();

  #define YYDEBUG 1

%}
...
```

2. Dar a la variable `yydebug` un valor distinto a 0 en alguna parte del código C del analizador, por ejemplo en el programa principal.

```
...
int main(int argc, char *argv[]) {

  yydebug = 1;

  if (argc < 2) {
    printf("Uso: ./c-dull NombreArchivo\n");
    }
  else {
    yyin = fopen(argv[1],"r");
    yyparse();
    }
  }
...
```

Activando la macro `YYDEBUG`, el analizador sintáctico listará por la consola, a medida que realiza el análisis de la entrada, los tokens que va leyendo de yylex(), las reglas que va reduciendo, los estados por los que va transitando y el contenido de la pila del analizador.

3.11. Ejemplo

Os he dejado un código de ejemplo (`prueba.c`) en el archivo `c-dull.tar.gz`. El código puede ser analizado sintácticamente usando una gramática que concuerde con la especificación dada en este enunciado, pero no tiene mucho sentido semánticamente. Es una colección de declaraciones diseñada para probar el funcionamiento de las diferentes partes de la gramática (pero no necesariamente todas).

Recordad que la salida del analizador tiene que ser un volcado de los tokens que se van leyendo y de las reglas que se van reduciendo. El resultado de aplicar vuestro analizador a `ordenar.c`, debería ser parecido a esto:

```
linea 1, palabra reservada: using
linea 1, identificador: MATH
linea 1, operador =
linea 1, identificador: math
linea 1, delimitador: ;
  id_tipos -> ID
  nom_tipo_o_esp_noms -> id_tipos
  dir_uso -> USING ID = nom_tipo_o_esp_noms
  lista_dir_uso -> dir_uso
linea 2, palabra reservada: using
linea 2, identificador: STDIO
```

```
linea 2, operador =
linea 2, identificador: stdio
linea 2, delimitador: ;
  id_tipos -> ID
  nom_tipo_o_esp_noms -> id_tipos
  dir_uso -> USING ID = nom_tipo_o_esp_noms
  lista_dir_uso -> lista_dir_uso dir_uso
linea 4, palabra reservada: unsigned
  signo -> UNSIGNED
linea 4, palabra reservada: short
  long -> SHORT
linea 4, palabra reservada: int
  tipo_basico -> INT
  tipo_escalar -> signo long tipo_basico
  tipo -> tipo_escalar
linea 4, identificador: prueba

...

  expr_and -> expr_or_bin
  expr_logica -> expr_and
  expr -> expr_logica
  expr_par -> (expr)
  expr_pos -> expr_par
linea 123, delimitador: ;
  expr_pref -> expr_pos
  expr_cast -> expr_pref
  expr_mult -> expr_cast
  expr_add -> expr_mult
  expr_despl -> expr_add
  expr_and_bin -> expr_despl
  expr_xor_bin -> expr_and_bin
  expr_or_bin -> expr_xor_bin
  expr_rel -> expr_or_bin
  expr_eq -> expr_rel
  expr_and -> expr_or_bin
  expr_logica -> expr_and
  expr -> expr_logica
  instr_return -> RETURN expr ;
  instr -> instr_return
  list_instr -> list_instr instr
linea 124, delimitador: }
  blq_instr -> { list_instr }
  declr_func -> firm_func blq_instr
  decl -> decl_func
  list_decl -> list_decl decl
linea 125, delimitador: }
  bloq_esp_nom -> { lista_decl }
  decl_esp_nom -> NMSPACE id_anid bloq_esp_nom
  decl -> decl_esp_nom
  list_decl -> list_decl decl
  modulo -> lista_dir_uso list_decl
```

A. Definición (casi) completa de la gramática de C-Dull

Para que no tengáis que ir buscando cacho a cacho en el texto. Obviamente, faltan las definiciones de las reglas para operadores binarios en las expresiones, dado que sólo los he enumerado con sus precedencias y asociatividades.

```
****************************************PROGRAMA****************************************
  modulo ::= [ directiva_uso ]* [ declaracion ]+

  declaracion ::= declaracion_espacio_nombres
                | declaracion_variables
                | declaracion_tipo
                | declaracion_funcion

  directiva_uso ::= USING [ IDENTIFICADOR '=' ]? nombre_tipo_o_espacio_nombres ';'

  nombre_tipo_o_espacio_nombres ::= [ identificador_con_tipos '.' ]* identificador_con_tipos

  identificador_con_tipos ::= IDENTIFICADOR [ '(' ( nombre_tipo_o_espacio_nombres )+ ')' ]?

**********************************ESPACIO DE NOMBRES**********************************
  declaracion_espacio_nombres ::= 'namespace' identificador_anidado bloque_espacio_nombres

  identificador_anidado ::= [ IDENTIFICADOR '.' ]* IDENTIFICADOR

  bloque_espacio_nombres ::= '{' [ directiva_uso ]* [ declaracion ]+ '}'

**************************************VARIABLES**************************************
  declaracion_variable ::= tipo ( nombre )+ ';'

  tipo ::= '<' nombre_tipo_o_espacio_nombres '>'
         | tipo_escalar

  tipo_escalar ::= [ signo ]? [ longitud ]? tipo_basico

  longitud ::= SHORT | LONG

  signo ::= SIGNED | UNSIGNED

  tipo_basico ::= CHAR | INT | FLOAT | DOUBLE | BOOLEAN

  nombre ::= dato [ '=' valor ]?

  dato ::= [ '*' ]* dato_indexado

  dato_indexado ::= IDENTIFICADOR [ '[' ( expresion )* ']' ]*

  valor ::= expresion
          | '{' ( valor )+ '}'

****************************************TIPOS****************************************
  declaracion_tipo ::= nombramiento_tipo
                     | declaracion_struct_union
                     | declaracion_interfaz
                     | declaracion_enum
                     | declaracion_clase

  nombramiento_tipo ::= 'typedef' tipo ID ';'
```

```
  declaracion_struct_union ::= [ modificador ]* struct_union [ IDENTIFICADOR ]?
                               '{' [ declaracion_campo ]+ '}'

  modificador ::= 'new' | 'public' | 'protected' | 'internal' | 'private' | 'static'
                | 'virtual' | 'sealed' | 'override' | 'abstract' | 'extern'

  struct_union ::= 'struct' | 'union'

  declaracion_campo ::= tipo ( nombre )+ ';'
                      | declaracion_struct_union ( nombre )+ ';'

  declaracion_interfaz ::= [ modificador ]* 'interface' IDENTIFICADOR herencia cuerpo_interfaz

  herencia ::= [ ':' ( nombre_tipo_o_espacio_nombres )+ ]?

  cuerpo_interfaz ::= '{' [ declaracion_metodo_interfaz ]* '}'

  declaracion_metodo_interfaz ::= [ 'new' ]? firma_funcion ';'

  declaracion_enum ::= [ modificador ]* 'enum' IDENTIFICADOR [ ':' tipo_escalar ]? cuerpo_enum

  cuerpo_enum ::= '{' ( declaracion_miembro_enum )+ '}'

  declaracion_miembro_enum ::= IDENTIFICADOR [ '=' expresion ]?

****************************************CLASES****************************************
  declaracion_clase ::= [ modificador ]* 'class' IDENTIFICADOR herencia cuerpo_clase

  cuerpo clase ::= '{' [ declaracion_elemento_clase ]+ '}'

  declaracion_elemento_clase ::= declaracion_tipo
                               | declaracion_atributo
                               | declaracion_metodo
                               | declaracion_constructor
                               | declaracion_destructor
                               | declaracion_atributo

  declaracion_atributo ::= [ modificador ]* declaracion_variable

  declaracion_metodo ::= [ modificador ]* firma_funcion bloque_instrucciones

  declaracion_constructor ::= [ modificador ]* cabecera_constructor bloque_instrucciones

  cabecera_constructor ::= IDENTIFICADOR [ parametros ]? [ inicializador_constructor ]?

  inicializador_constructor ::= ':' BASE parametros
                              | ':' THIS parametros

  declaracion_destructor ::= [ modificador ]* cabecera_destructor bloque_instrucciones

  cabecera_destructor ::= '~' IDENTIFICADOR '(' ')'

*************************************FUNCIONES*************************************
  declaracion_funcion ::= firma_funcion bloque_instrucciones
```

```
  firma_funcion ::= VOID IDENTIFICADOR parametros
                  | tipo [ '*' ]* IDENTIFICADOR parametros

  parametros ::= '(' [ argumentos ';' ]* [ argumentos ]? ')'

  argumentos ::= nombre_tipo ( variable )+

  nombre_tipo ::= tipo [ '*' ]*

  variable ::= IDENTIFICADOR [ '=' expresion ]?

**********************************INSTRUCCIONES*****************************************
  instruccion ::= bloque_instrucciones
                | instruccion_expresion
                | instruccion_bifurcacion
                | instruccion_bucle
                | instruccion_salto
                | instruccion_destino_salto
                | instruccion_retorno
                | instruccion_lanzamiento_excepcion
                | instruccion_captura_excepcion
                | instruccion_vacia

  bloque_instrucciones ::= '{' [ declaracion ]* [ instruccion ]* '}'
  instruccion_expresion ::= expresion_funcional ';' | asignacion ';'

  asignacion ::= expresion_indexada operador_asignacion expresion

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

  instruccion_bifurcacion ::= 'if' '(' expresion ')' instruccion [ 'else' instruccion ]?
                            | 'switch' '(' expresion ')' '{' [ instruccion_caso ]+ '}'

  instruccion_caso ::= 'case' expresion ':' instruccion
                     | 'default' ':' instruccion

  instruccion_bucle ::= 'while' '(' expresion ')' instruccion
                      | 'do' instruccion 'while' '( expresion ')' ';'
                      | 'for' '(' ( asignacion )* ';' expresion ';' ( expresion )+ ')' instruccion

  instruccion_salto ::= 'goto' IDENTIFICADOR ';' | 'continue' ';' | 'break' ';'

  instruccion_destino_salto ::= IDENTIFICADOR ':' instruccion ';'

  instruccion_retorno ::= 'return' [ expresion ]? ';'

  instruccion_lanzamiento_excepcion ::= 'throw' expresion ';'

  instruccion_captura_excepcion ::= 'try' bloque_instrucciones clausulas-catch
                                  | 'try' bloque_instrucciones clausula_finally
                                  | 'try' bloque_instrucciones clausulas-catch clausula_finally

  clausulas-catch ::= [ clausula_catch_especifica ]+
                    | clausula_catch_general
                    | [ clausula_catch_especifica ]+ clausula_catch_general

  clausula_catch_especifica ::= 'catch' '(' nombre_tipo ')' bloque_instrucciones
```

```
  clausula_catch_general ::= 'catch' bloque_instrucciones

  clausula-finally ::= 'finally' bloque_instrucciones

  instruccion_retorno ::= 'return' [ expresion ]? ';'

*******************************EXPRESIONES (NO COMPLETAS)*******************************
  expresion_constante ::= ENTERO | REAL | CADENA | CARACTER | BOOLEANO

  expresion_parentesis ::= '(' expresion ')'

  expresion_funcional ::= identificador_anidado '(' ( expresion )* ')'

  expresion_creacion_objeto ::= 'new' identificador_anidado '(' ( expresion )* ')'

  expresion_indexada ::= identificador_anidado
                       | expresion_indexada '[' expresion ']'
                       | expresion_indexada '->' identificador_anidado

  expresion_postfija ::= expresion_constante
                       | expresion_parentesis
                       | expresion_funcional
                       | expresion_creacion_objeto
                       | expresion_indexada
                       | expresion_postfija '++'
                       | expresion_postfija '--'

  expresion_prefija ::= expresion_postfija
                      | 'sizeof' expresion_prefija
                      | 'sizeof' '(' nombre_tipo ')'
                      | operador_prefijo expresion_cast

  operador_prefijo ::= '++' | '--' | '&' | '*' | '+' | '-' | '~' | '!'

  expresion_cast ::= expresion_prefija
                   | '(' nombre_tipo ')' expresion_prefija
```