Practica 2.- Analizador Sintáctico de C--

TALF 2019/20

16 de junio de 2020

Índice

1. Descripción de la práctica

Objetivo: El alumno deberá implementar un analizador sintáctico en Bison para una versión reducida (y con algunos cambios) de C, a la que llamaremos C--. 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.

Ej.- DarribaBilbao-FernandezGonzalez

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.

[1] Pista: dangling else.

Ej.- DarribaBilbao-FernandezGonzalez.zip

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

Fecha de entrega: El código se subirá a Faitic como muy tarde el 22 de junio de 2020, a las 23:59.

Material: He dejado en Faitic (TALF → documentos y enlaces → material de prácticas) un fichero (`cminus2.tar.gz`). Dicho fichero contiene la especificación incompleta[2] en Flex del analizador léxico (`cminus2.l`), la especificación incompleta[3] del analizador sintáctico en Bison (`cminus2.y`), un fichero de ejemplo (`prueba.c--`), y un Makefile.

Nota máxima: ~~2'5~~ 3'75 ptos. Se evaluará al alumno por las partes del analizador que se hayan hecho satisfactoriamente:

- ~~0'7~~ 1'05 por el programa, declaraciones, macros y definición de funciones.
- ~~0'7~~ 1'05 puntos por las instrucciones.
- ~~0'7~~ 1'05 por las expresiones.
- ~~0'4~~ 0'6 por el tratamiento de errores.

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

2. Analizador léxico

Teenis que utilizar el analizador definido en la práctica 1. La única modificación a mayores, consiste en 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. Los nombres de los tokens (categorías léxicas) están definidos en `cminus2.y`, y teneis que utilizarlos.

- 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 '`real`', el nombre de token correspondiente será `REAL`, para '`goto`' será `GOTO`, etc.
- `SIZEOF` corresponde al operador '`sizeof`'.
- `CARACTER` corresponde a un carácter (incluyendo las comillas simples).
- `STRING` corresponde a una cadena (incluyendo las comillas dobles).
- `IDENTIFICADOR` corresponde a un identificador.
- `ENTERO` corresponde a un número entero.
- `REAL` corresponde a un número real.
- `PATH` es el numbre de archivo o path al mismo acompañando a un `#include` (delimitado por `< >`).
- Operadores:

```
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 '->'  POTENCIA '**'
```

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

3. Gramática del analizador sintáctico de C--

En esta sección vamos a proporcionar la especificación de la gramática de C--. 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_constante ::= '(' expresion ')'
  ```

  `expresion` y `expresion_constante` 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 | STRING | CARACTER | '(' expresion ')'
  ```

  que también puede escribirse como:

  ```
  expresion_constante ::= ENTERO
                        | REAL
                        | STRING
                        | CARACTER
                        | '(' expresion ')'
  ```

- 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:

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

  tenemos que un bloque de intrucciones está formada por una llave de apertura '`{`', 0 o más declaraciones, 0 o más instrucciones, y una llave de cierre '`}`'.
- 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:

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

  especifica que una `elementos` deriva 1 o más `elementos` entre llaves (de apertura y cierre).
- Los símbolos `[ ]?` delimitan los elementos opcionales. En el siguiente ejemplo:

  ```
  declaracion ::= declaracion_tipo ( nombre )* [ '#' ]? ';'
  ```

  '`#`' es opcional, por lo que puede aparecer o no en una declaración.

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

3.1. Bloques constituyentes de un programa C--

Un programa C-- está compuesto por una o más declaraciones, funciones o macros.

```
programa ::= [ bloque ]+

bloque ::= definicion_funcion
         | declaracion
         | macros
```

Una función se define declarando su tipo, el nombre de la función y las instrucciones que componen su código fuente. Una diferencia entre C y C-- es que, en este último, las argumentos de la función se declaran dentro del código de la misma, en `bloque_instrucciones`.

```
definicion_funcion ::= [ declaracion_tipo [ '*' ]* ]? IDENTIFICADOR bloque_instrucciones

macros ::= '#' 'include' PATH
         | '#' 'define' IDENTIFICADOR macro

macro ::= ENTERO | REAL | STRING | CARACTER | instruccion
```

Respecto a las macros, tenemos dos posibilidades: 'include' o 'define'. En este última, podemos asociar valores constantes (enteros, reales, caracteres o cadenas) o instrucciones al nombre.

3.2. Declaraciones

Una declaración nos permite definir tipos de datos o variables. En el primer caso, asociaremos un nombre a un tipo con la palabra reservada '`typedef`', mientras que una definición de variables está formada por una declaración de tipo y una lista de nombres de variables que tendrán ese tipo. El carácter '`#`' se emplea para señalar, dentro de una función, que la declaración actual es de uno o varios parámetros de la misma.

```
declaracion ::= declaracion_tipo ( nombre )* [ '#' ]? ';'
              | 'typedef' declaracion_tipo IDENTIFICADOR ';'
```

Tal y como está especificada, la definición de variables, en su forma más simple, está compuesta sólo de una declaracion de tipo seguida de un punto y coma. Ello es debido a que la definición un tipo de dato complejo (`struct` o `union`), con un nombre asociado, es una de las posibilidades dentro de una declaración de tipo, y no tiene por qué estar acompañada de una lista de nombres. Por ejemplo:

```
struct my_struct {
  int un_campo;
  real otro_campo:
  }; // no hay lista de nombres aqui
```

Por supuesto, esta definición de declaración tiene un problema de sobregeneración (la gramática puede derivar cosas que no debería), como:

```
int;
```

La razón de que la gramática se haya escrito así, es que sea lo más compacta y legible posible. En lugar de tener una gramática un poco más complicada que elimine este problema, retrasamos la resolución del mismo a la fase de análisis semántico, y, en particular, al chequeo de tipos. De este modo, podríamos implementar las acciones semánticas de las reglas asociadas a esta parte de la gramática de manera que, en el caso de que no haya una lista de nombres acompañando a `declaracion_tipo`, el compilador notifique un error si el tipo asociado a dicha declaración no corresponde a una `struct` o `union`.

Lo que teneis que hacer en este caso, dado que no habeis visto nada en clase sobre análisis semántico, es implementar las reglas de esta especificación, suponiendo que el problema anterior se resolvería posteriormente durante el chequeo de tipos.

En otro orden de cosas, las declaraciones de tipo permiten especificar un tipo básico (con los modificadores '`extern`', '`static`', '`auto`', '`register`', '`short`' o '`long`' y '`signed`' o '`unsigned`'), o un tipo de dato complejo (`struct`, `union` o `enum`).

```
declaracion_tipo ::= [ almacenamiento ]* tipo_basico_modificado
                   | [ almacenamiento ]* definicion_struct_union
                   | [ almacenamiento ]* definicion_enum

tipo_basico_modificado ::= [ signo ]? [ longitud ]? tipo_basico
                         | '[' IDENTIFICADOR ']'

almacenamiento ::= 'extern' | 'static' | 'auto' | 'register'

longitud ::= 'short' | 'long'

signo ::= 'signed' | 'unsigned'

tipo_basico ::= 'void' | 'char' | 'int' | 'float' | 'double'
```

Otra diferencia entre C-- y C está en el uso de tipos de datos predefinidos. En la regla:

```
tipo_basico_modificado ::= '[' IDENTIFICADOR ']'
```

el identificador es el nombre de un tipo de dato definido anteriormente en el programa. Lo que suelen hacer los analizadores léxicos de los compiladores de C, cada vez que encuentra un identificador, es comprobar si se trata de un tipo de dato, variable o constante, 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, lo que daría lugar a conflictos en el analizador sintáctico. Para prevenirlos, cuando en C-- se define una variable usando un tipo definido previamente, el nombre de tipo se pone entre corchetes. 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`.

Los tipos de datos complejos se declaran de la misma manera, ya sean structs o unions. Por un lado, podemos definir el tipo de dato complejo, dándole opcionalmente un nombre, y listando los campos que lo componen entre llaves (a través de `declaracion_struct`). Otra posibilidad es declarar una o más variables con un tipo complejo declarado previamente.

```
definicion_struct_union ::= struct_union [ IDENTIFICADOR ]? '{' [ declaracion_struct ]+ '}'
                          | struct_union IDENTIFICADOR

struct_union ::= 'struct' | 'union'

declaracion_struct ::= tipo_basico_modificado ( nombre )+ ';'
                     | definicion_struct_union ( nombre )+ ';'
```

Finalmente, enumeramos la lista de nombres (de variables) que se van a definir, perteneciendo al tipo de dato definido previamente en la declaración, o a un puntero al mismo. También podemos asignar valores iniciales a dichas variables. En el caso de datos escalares, el valor será el resultado de una expresión (descritas en el siguiente apartado), mientras que en el caso de arrays, definiremos sus elementos entre llaves y separados por comas.

```
nombre ::= dato [ '=' elementos ]?

dato ::= [ '*' ]* IDENTIFICADOR [ '[' [ expresion ]? ']' ]*

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

Por su parte, 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 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 de la definición del tipo.

```
definicion_enum ::= 'enum' IDENTIFICADOR [ ':' tipo_basico_modificado ]? 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_basico_modificado` declarado tras el nombre de tipo enumerado [6]. Por ejemplo:

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

3.3. 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
              | ';'
```

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 expresión está formada bien por una llamada a función o bien por una asignación. Dicha asignación está, a su vez, compuesta por una expresión indexada seguida de un operador de asignación y una expresión.

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

asignacion ::= expresion_indexada operador_asignacion expresion

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

[6] Esa comprobación de tipos se haría en el análisis semántico, así que podéis ignorarla.

Las instrucciones de bifurcación son de dos tipos: if-else y switch-case-default. En el primero de ellos, tenemos un token '`if`', seguido de una expresion (la condición del if-then-else) entre paréntesis, y una instrucción. Opcionalmente, podemos tener un '`else`' seguido de una instrucción. En ambos casos, se finaliza con un nuevo '`if`'. Recordad que `instruccion` puede ser, a su vez, un bloque de instrucciones.

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

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

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 token '`:`' 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' '(' ( definicion_asignacion )* ';' expresion ';' expresion ')'
                      instruccion
                    | 'for' '(' [ declaracion_tipo [ '*' ]* ]? IDENTIFICADOR ';'
                      expresion ';' sentido ')' instruccion

definicion_asignacion ::= asignacion
                        | declaracion_tipo [ '*' ]* expresion_indexada '=' expresion

sentido ::= '--'
          | '++'
```

En el bucle for se pueden definir (incluyendo su tipo) e inicializar los valores de las variables indice del bucle, separando dichas asignaciones por comas, siempre y cuando no se hayan realizado anteriormente en el código. También consideramos (como variante adicional del bucle `for`) un tipo de bucle que suele existir en algunos lenguajes de programación como `foreach`. En ese segundo tipo de bucle, definimos una variable en la que vamos almacenando el valor de una posición del array en cada iteración, y a continuación la `expresion` en la que se define dicho array (generalmente el nombre del mismo), más un operador indicando el sentido ascendente o descendente en el que se recorre el bucle.

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.

Finalmente, 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.

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

3.4. Expresiones

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

```
expresion_constante ::= ENTERO | REAL | STRING | CARACTER

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

Aquí tenemos otro 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, de nuevo podeis implementar la especificación tal como está.

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

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

expresion_indexada ::= IDENTIFICADOR
                     | expresion_indexada '[' expresion ']'
                     | expresion_indexada '.' IDENTIFICADOR
                     | expresion_indexada '->' IDENTIFICADOR
```

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_indexada
                     | expresion_postfija '++'
                     | expresion_postfija '--'

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

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

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

nombre_tipo ::= tipo_basico_modificado [ '*' ]*
```

A partir de aquí, teneis 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:

- '**' (potencia)
- '*', '/' 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, excepto en el caso de la potencia ('**'), para la que es por la derecha.

A la hora de diseñar las reglas para los operadores binarios, podeis implementar la precedencia y asociatividad diseñando una gramática determinista, o podeis 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 implementeis esta porción de la especificación como una gramática determinista. Si lo seguís, podeis 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:

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

También debeis 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 todos los operadores, si llamamos al simbolo raíz de la porción de la gramática correspondiente a todas los operadores y operandos anteriores `expresion_logica`, podemos definir una `expresion` como:

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

3.5. 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-- 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 y macros.
4. Bloques y Declaraciones (definiciones de tipos de datos y variables).

Este procedimiento tiene la ventaja de que, si no os da tiempo a implementar toda la gramática, podeis presentarme una porción de la misma que funcione.

3.6. Depuración de la gramática

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

1. Declarar dicha macro en la sección de declaraciones de Bison.

```
%{

  #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: ./cminus2 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.

3.7. Ejemplo

Os he dejado un programa de ejemplo (`prueba.c--`) en el archivo `cminus2.tar.gz`. 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.stu`, debería ser parecido a esto:

```
linea 1, operador #
linea 1, palabra reservada: include
linea 1, path: <math.h>
  macros -> #INCL PATH
  blq -> macros
  blqs -> blk
linea 2, operador #
linea 2, palabra reservada: include
linea 2, path: <stdio.h>
```

```
  macros -> #INCL PATH
  blq -> macros
  blqs -> blqs blq
linea 4, palabra reservada: unsigned
  signo -> UNSIGNED
linea 4, palabra reservada: short
  long -> SHORT
linea 4, palabra reservada: int
  tipo_basico -> INT
  tipo_basico_mod -> signo long tipo_basico
  decl_tipo -> tipo_basico_mod
linea 4, identificador: prueba
linea 4, delimitador: ;
  dato_index -> ID
  dato -> dato_index
  nom -> dato
  list_noms -> nom
  decl -> decl_tipo list_noms
  blq -> decl
  blqs -> blqs blq

...

linea 76, delimitador: ;
  expr_pref -> expr_pos
  expr_cast -> expr_pref
  expr_pot -> expr_cast
  expr_mult -> expr_pot
  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 77, delimitador: }
  blq_instr -> { list_decl list_instr }
  def_func -> def_tipo ID blq_instr
  blq -> def_func
  blqs -> blqs blq
  EXITO: prog -> blks
```

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

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****************************************

  programa ::= [ bloque ]+

  bloque ::= definicion_funcion
           | declaracion
           | macros

  definicion_funcion ::= [ declaracion_tipo [ '*' ]* ]? IDENTIFICADOR bloque_instrucciones

  macros ::= '#' 'include' PATH
           | '#' 'define' IDENTIFICADOR macro

  macro ::= ENTERO | REAL | STRING | CARACTER | instruccion

************************************DECLARACIONES************************************

  declaracion ::= declaracion_tipo ( nombre )* [ '#' ]? ';'
                | 'typedef' declaracion_tipo IDENTIFICADOR ';'

  declaracion_tipo ::= [ almacenamiento ]* tipo_basico_modificado
                     | [ almacenamiento ]* definicion_struct_union
                     | [ almacenamiento ]* definicion_enum

  tipo_basico_modificado ::= [ signo ]? [ longitud ]? tipo_basico
                           | '[' IDENTIFICADOR ']'

  almacenamiento ::= 'extern' | 'static' | 'auto' | 'register'

  longitud ::= 'short' | 'long'

  signo ::= 'signed' | 'unsigned'

  tipo_basico ::= 'void' | 'char' | 'int' | 'float' | 'double'

  definicion_struct_union ::= struct_union [ IDENTIFICADOR ]? '{' [ declaracion_struct ]+ '}'
                            | struct_union IDENTIFICADOR

  struct_union ::= 'struct' | 'union'

  declaracion_struct ::= tipo_basico_modificado ( nombre )+ ';'
                       | definicion_struct_union ( nombre )+ ';'

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

  dato ::= [ '*' ]* IDENTIFICADOR [ '[' [ expresion ]? ']' ]*

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

  definicion_enum ::= 'enum' IDENTIFICADOR [ ':' tipo_basico_modificado ]? cuerpo_enum

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

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

************************************INSTRUCCIONES*************************************

  instruccion ::= bloque_instrucciones
                | instruccion_expresion
                | instruccion_bifurcacion
                | instruccion_bucle
                | instruccion_salto
                | instruccion_destino_salto
                | instruccion_retorno
                | ';'

  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' '(' ( definicion_asignacion )* ';' expresion ';' expresion ')'
                        instruccion
                      | 'for' '(' [ declaracion_tipo [ '*' ]* ]? IDENTIFICADOR ';'
                        expresion ';' sentido ')' instruccion

  definicion_asignacion ::= asignacion
                          | declaracion_tipo [ '*' ]* expresion_indexada '=' expresion

  sentido ::= '--'
            | '++'

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

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

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

*******************************EXPRESIONES (INCOMPLETAS)*******************************

  expresion_constante ::= ENTERO | REAL | STRING | CARACTER

  expresion_parentesis ::= '(' expresion ')'


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

```
expresion_indexada ::= IDENTIFICADOR
                     | expresion_indexada '[' expresion ']'
                     | expresion_indexada '.' IDENTIFICADOR
                     | expresion_indexada '->' IDENTIFICADOR

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

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

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

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

nombre_tipo ::= tipo_basico_modificado [ '*' ]*
```