Profesor: EDUARDO MARTINEZ GRACIA

Compiladores

Práctica Curso 16/17

Convocatoria de Junio

Componentes:

Carlos Antonio López Córdoba cantonio.lopez@um.es 77757141

José Andrés Ríos Gómez joseandres.rios@um.es 48651079

ÍNDICE

INTRODUCCIÓN

Esta práctica tiene como objetivo estudiar e implementar un compilador(MiniC). Para ello haremos uso de diferentes herramientas cuyo material y tutoriales son proporcionados por los profesores de la asignatura.

Nosotros hemos implementado sólo los módulos obligatorios para aprobar la asignatura que son el analizador léxico y el sintáctico.

Para el análisis léxico hacemos uso de la herramienta Flex, la cual se puede usar tanto en Windows como en Linux. En este caso nosotros hemos hecho todo en Linux.

Para el análisis sintáctico hacemos uso de la herramienta Bison combinada con Flex, una vez hecho el léxico.

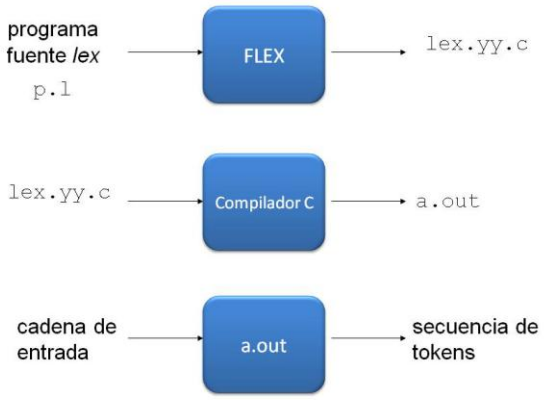
Como mejoras de la práctica hemos ampliado el lenguaje con nuevos tipos como Float o Boolean.

ANÁLISIS LÉXICO CON FLEX

Flex es una herramienta para generar analizadores léxicos a partir de un fichero que contenga especificaciones de expresiones regulares y definidas por el usuario.

A partir de un fichero, que tiene extensión .l y que ahora explicaremos, genera otro llamado lex.yy.c, que después de compilarse funcionan como analizador léxico.

El siguiente esquema muestra mejor la creación del analizador léxico:



Ahora explicaremos la estructura de nuestro fichero .l, que en nuestro caso se llama minic.l.

El formato de cualquier .l es el siguiente:

Formato de un fichero.l :
{definiciones}
%%
{reglas}
%%
{subrutinas de usuario}

En el apartado de definiciones se especifican macros que dan nombres a expresiones regulares auxiliares que serán utilizadas en la sección de reglas más adelante. También se puede incluir código C. En nuestro caso el apartado de definiciones es el siguiente:

```
%{
#include "minic.tab.h"
#include <stdio.h>

int comienzo_comentario;
%}

L           [a-zA-Z]
D           [0-9]
panico      [^a-zA-Z_0-9\";,+\-*/=(){}<>!&| \t\n\r]
ok          [a-zA-Z_0-9\";,+\-*/=(){}<>! \t\n\r]

%x comentario
%x comentario_multilinea
%x cadena
%x andlogico
%x orlogico
```

Por ejemplo, declaramos las expr. regulares auxiliares L y D que representan a cualquier letra y cualquier dígito respectivamente y que usarémos en el siguiente apartado del .l.

También hacemos uso de la expr. regular pánico que representan cualquier token desconocido para nuestro analizador personal y que nos servirá para tratar el modo pánico más adelante.

Por último declaramos las condiciones de arranque o de contexto que usaremos y que van precedidas de %x como podemos observar. En este caso son exclusivas y serán activadas cuando el intérprete del autómata se encuentra en una condición de entrada de este tipo.

En nuestro caso tenemos 5, y trataremos los comentarios, comentarios multilínea, las cadenas, el OR y AND lógico.

En el apartado de las reglas es donde según la expresión regular que tengamos ejecutaremos una sentencia en C según el lexema asociado a dicha expresión regular. La siguiente tabla muestra las expresiones regulares básicas(palabras reservadas y operadores sobre todo) de nuestro analizador y su acción. Todos los “return” lo que devuelven realmente es un entero y esta equivalencia de esta definida en el fichero.h, en nuestro caso minic.h.

EXPR. REGULAR	ACCIÓN
func	return FUNC;
var	return VAR;
let	return LET;
float	return FLOAT;
if	return IF;
else	return ELSE;
while	return WHILE;
print	return PRINT;
read	return READ;
bool	return BOOL;
true	return TRUE;
false	return FALSE;
({L}|_)({L}|_|{D})*	return ID;
{D}+	Return NUM;
{D}+\.{D}+	return DECIMAL;
"<="	return LESSEQ;
">="	return GREATEREQ;
"=="	return EQ;
"!="	return NOTEQ;
"<"	return LESS;
">"	return GREATER;
"!"	return NOT;
";"	return PYC;
","	return COMMA;
"+"	return PLUSOP;
"-"	return MINUSOP;
"*"	return MULTOP;
"/"	return DIVOP;
"="	return ASIGN;
"("	return PARI;
")"	return PARD;
"{"	return LLAVEI;
"}"	return LLAVED;

Cabe destacar algunos matices que no se reflejan en la tabla como por ejemplo los que se muestran en la siguiente imagen de nuestro .l

```
({L}|_)({L}|_|{D})*      { if (yyleng > 16) {
                             printf("Error: ID con mas de 16 caracteres en línea %d: %s (%lu caracteres)\n",yylineno,yytext,yyleng);
                           } else {
                             yylval.str = strdup(yytext);
                           } return ID; }
{D}+                     { if (atol(yytext) > 2147483648) {
                               printf("Error: NUM provoca desbordamiento en línea %d: %s\n",yylineno,yytext);
                           } else {
                             yylval.num = atoi(yytext);
                           } return NUM; }
{D}+\.{D}+               {   yylval.dec = atof(yytext); return DECIMAL; }
```

Para los identificadores(ID), si tienen una longitud superior a 16 mostraremos un error.

Para los enteros con signo, si el valor se excede de 2^31, imprimiremos error de desbordamiento.

Por último comentaremos las condiciones de arranque o contexto que mencionamos anteriormente.

Para las cadenas, activaremos la condición <cadena> en cuanto el analizador encuentra unas comillas, por lo que traducido al lenguaje del analizador, para reconocer el lexema de unas comillas, al ser un carácter especial tendremos que ponerle la contrabarra: \”

Una vez reconocido activaremos la condición mediante BEGIN(cadena). En cuanto vuelva a reconocer el mismo lexema que indicará que la cadena ha terminado(\”) o un EOF o un \n entonces volvemos al estado original mediante BEGIN(INITIAL).

Para los OR y los AND el procedimiento es el mismo, si reconoce el lexema “|” o “&” haremos BEGIN(orlogico) o BEGIN(andlogico) respectivamente. Al necesitar dos caracteres iguales seguidos, si lo encuentra volveremos al estado original mediante BEGIN(INITIAL) como siempre y si no también volveremos al estado original pero imprimiendo un error léxico.

Para los comentarios normales simplemente usaremos BEGIN(comentario) cuando reconozca el lexema de los comentarios de una línea “//” y volveremos al estado inicial cuando se encuentre un retorno de carro.

Para los comentarios multilínea usaremos BEGIN(comentario_multilinea) cuando reconozca el lexema que marca el comienzo de un comentario “/*” y volveremos al estado inicial cuando reconozca el lexema que marca el final de un comentario “*/” o un EOF, pero en este último caso mostrando error. Para reconocer el interior del comentario tenemos una expresión regular que describe las cadenas de caracteres que no contengan la subcadena “*/”.

En la siguiente imagen se muestra lo comentado justo arriba:

```
\"                                  { BEGIN(cadena); yymore(); }
<cadena>\"                          { BEGIN(INITIAL); yylval.str = strdup(yytext); return STRING; }
<cadena>[^"\n]+                     { yymore(); }
<cadena>"\n"                        { printf("Cadena abierta en línea %d sin cerrar\n", yylineno);  BEGIN(INITIAL); }
<cadena><<EOF>>                     { printf("Cadena abierta al final del documento sin cerrar\n");  return 0; }

"&"                                 { BEGIN(andlogico); yymore(); }
<andlogico>"&"                      { BEGIN(INITIAL); return AND; }
<andlogico>{ok}+                    { BEGIN(INITIAL); printf("Error léxico (línea %d): &\n",yylineno); yyless(1); }


"|"                                 { BEGIN(orlogico); yymore(); }
<orlogico>"|"                       { BEGIN(INITIAL); return OR; }
<orlogico>{ok}+                     { BEGIN(INITIAL); printf("Error léxico (línea %d): |\n",yylineno); yyless(1); }

"//"                                { BEGIN(comentario); }
<comentario>"\n"                    { BEGIN(INITIAL); }
<comentario>[^\n]                   { }

"/*"                                    { BEGIN(comentario_multilinea); comienzo_comentario = yylineno; yymore(); }
<comentario_multilinea>([^*]|[*]^+[^*/])*   { yymore(); }
<comentario_multilinea>"*/"             { BEGIN(INITIAL); }
<comentario_multilinea><<EOF>>          { printf("Comentario abierto en línea %d sin cerrar\n", comienzo_comentario); return 0; }

[ \t\n\r]+                          { /* Ignorar espacios en blanco, tabuladores y otros delimitadores */ }
{panico}+                           { /* Error léxico */ printf("Error léxico (línea %d): %s\n",yylineno,yytext); }
```

Las dos últimas líneas son las que usamos para ignorar los espacios en blanco, tabuladores y demás y para tratar el modo pánico.

ANÁLISIS SINTÁCTICO

Una vez realizado el análisis léxico de nuestra entrada y reconocidos los tokens que componen nuestra gramática, es necesario reconocer que nuestro fichero es sintácticamente correcto.

Para ello, usaremos Bison, que nos permite crear las reglas necesarias para el buen funcionamiento de nuestro compilador, además de definir más opciones que permitirán poder dar solución a los conflictos desplaza / reduce y a las expresiones ambiguas.

Para empezar, las reglas semánticas usadas en la práctica son las mismas que se proporcionan en el enunciado de prácticas, en el apartado 2.1.2 *Una sintaxis para miniC*. Pero, al hacer uso de una gramática extendida, es necesario añadir nuevas reglas, o modificar ya existentes, las cuales pasan a describirse a continuación:

1ª	declarations	-> declarations float identifier_list_float ;
2ª	identifier_list_float	-> asig_float ;
3ª		| identifier_list_float , asig_float
4ª	asig_float	-> id
5ª		| id = expression
6ª	statement	-> if (expression_comparacion) statement else statement
7ª		| if (expression_comparacion) statement
8ª		| while (expression_comparacion) statement
9ª	expression_comparacion	-> expression < expression
10ª		| expression > expression
11ª		| expression <= expression
12ª		| expression >= expression
13ª		| expression != expression
14ª		| expression == expression
15ª		| !expression_comparacion
16ª		| (expression_comparacion) && (expression_comparacion)
17ª		| (expression_comparacion) || (expression_comparacion)
18ª		| true
19ª		| false
20ª	expression	-> decimal

1ª: En nuestra gramática extendida vamos a permitir la declaración de un nuevo tipo, que es el float. Por ello, es necesario poder diferenciarlo de los enteros, por lo que se especifica una nueva regla únicamente para su declaración.

2ª y 3ª: Donde podremos realizar tanto una como todas las declaraciones de float que queramos.

4ª y 5ª: Donde se realizará la asignación de los float, bien por un identificador ya declarado previamente, o bien por una nueva expression.

6ª, 7ª y 8ª:: Al permitir booleanos en nuestra gramática, se puede refinar más el tratamiento de los if, if-else y los while, haciendo que la condición a tratar sea una expression_comparacion, que será de tipo booleano.

9ª a 19ª: Por ello, todas estas nuevas reglas darán a conocer el valor de comparaciones entre expresiones (si es mayor, menor, mayor o igual, si es menor o igual, es igual o no es igual, etc...), el valor opuesto de una expression_comparacion gracias al operando not, comparar dos expression_comparacion gracias al and y al or, o directamente, asignar un valor

true o false a una expression_comparacion. De esta manera, podemos ir sacando los valores booleanos que deseemos para el tratamiento de las reglas 6ª, 7ª y 8ª, como se ha mencionado anteriormente.

20ª: Anteriormente, dejábamos que un nuevo identificador expression sea asignado a un identificador float. La gramática sin extender permitía asignar un número entero a una expression, pero no un float. Por ello, se extiende para poder recuperar el token capturado por el analizador léxico y conocer el valor que tiene un decimal, de manera que una expression pueda ser de tipo float, con un valor decimal, además de permitir que haya expression de tipo entero, con un valor entero.

Teniendo en cuenta todas estas modificaciones y adicciones, las reglas de la gramática extendida pasa a ser en su conjunto:

```
program                 ->  func id ( ) { declarations statement_list }
declarations                ->  declarations var identifier_list;
                            |   declarations let identifier_list;
                            |   declarations float identifier_list_float;
identifier_list             ->  asig
                            |   identifier_list , asig
asig                        ->  asig
                            |   id = expression
identifier_list_float   ->  asig_float
                            |   identifier_list_float , asig_float
asig_float              ->  id
                            |   id = expression
statement_list              ->  statement_list statement
                            |   λ
statement               ->  id = expression ;
                            |   id = expression_float ;
                            |   id = expression_boolean ;
                            |   { statement_list }
                            |   if ( expression_comparacion ) statement else statement
                            |   if ( expression_comparacion ) statement
                            |   while ( expression_comparacion ) statement
                            |   print print_list ;
                            |   read read_list ;
print_list              ->  print_item
                            |   print_list , print_item
print_item              ->  expression
                            |   string
read_list               ->  id
                            |   read_list , id
expression_comparacion  ->  expression < expression
                            |   expression > expression
                            |   expression <= expression
                            |   expression >= expression
                            |   expression != expression
                            |   expression == expression
                            |   ! expression_comparacion
                            |   (expression_comparacion) && (expression_comparacion)
                            |   (expression_comparacion) || (expression_comparacion)
                            |   true
                            |   false
expression                  ->  expression + expression
                            |   expression - expression
                            |   expression * expression
                            |   expression / expression
                            |   - expression
                            |   ( expression )
                            |   id
                            |   num
                            |   decimal
```

Una vez definida las reglas, y reconocida la gramática, es necesario ver si es necesario resolver ambigüedades y/o conflictos.

En nuestro caso, la regla de producción que tratan sobre el if-else van a provocar un conflicto desplaza/reduce, al llegar a este punto:

```
if ( expression_comparacion ) statement · else statement
if ( expression_comparacion ) statement ·
```

Si no se realiza ningún tratamiento, nos saldrá un warning, pero el propio bison lo resuelve de manera correcta (desplazando). Pero nos interesa tenerlo todo bien resuelto, sin ningún warning, por lo que para ello, es necesario definir declarar dos tokens, NOELSE y ELSE como %nonassoc, haciendo que en la segunda regla, al final se incluya %prec NOELSE, de manera que podemos diferenciar una regla de la otra, y de esta manera, que no haya ningún conflicto.

Para la ambigüedad de precedencia a la hora de realizar operaciones, es necesario distinguir cuál son los tokens con mayor precendencia. Por ello, definiremos las precedencias para los operadores aritméticos de la siguiente manera, ya que se construyen en orden creciente de prioridad:

```
%left PLUSPOP MINUSOP (sumas y restas)
%left MULTOP DIVOP (multiplicaciones y divisiones)
```

A la hora de usar la regla:

```
expression -> -expression
```

es necesario avisar de que no se trata de una operación aritmética de resta, por lo que se define el token %nonassoc UMENOS, de manera que pueda reconocer una expression cuyo valor va a ser en negativo, indicando en la regla de producción que se usa dicho token con %prec al final de la regla.

Lo mismo ocurre para la regla de producción:

```
expression_comparacion -> !expression_comparacion
```

al encontrar el token !, por lo que sería necesario definirlo como %nonassoc, pero al ser la única regla de producción que usa dicho token (! como token en sí, no confundir con el token !=, ya que se compone además de =), no es necesario añadir a su regla de producción %prec.

Una vez hemos llegado a este paso, es necesario ver si algunos de nuestros tokens pueden ser de algún tipo específico.

Para ello, definimos un soporte para tipos de valores arbitrarios gracias a %union, que se compondrá de:

```
%union {
        int type;  //Tipo asociado a expression, con 0 para enteros, 1 para
                                //float, 2 para boolean, 3 para indicar que hay un error,
                                //explicado más adelante)
        int num;  //Valor asociado a un entero
        float dec;//Valor asociado a un float
        char *str;//Valor asociado a un identificador (nombre) o para
                                //un string (contenido)
}
```

Ahora toca identificar los tokens que van a tener un tipo asociado. Tras lo explicado anteriormente, nos quedan las siguientes asociaciones:

```
%token<num> NUM
%token<str> ID
%token<str> STRING
%token<dec> DECIMAL
```

Pero, ¿cómo podemos pasarle el valor a estos tokens? Para ello, en el analizador léxico vamos a hacer uso de yylval, el cual tiene los tipos definidos en la unión. Así, cuando se reconozca un token de los asociados anteriormente, en el yylval podremos asignar un valor para esos tipos, y recuperarlo en el analizador sintáctico.

Por ello, para los enteros, se usará yylval.num, para los decimales, yylval.dec, y para los identificadores y las cadenas, yylval.str.

Una vez asociado el tipo, nos interesa asociar, para las expression, el tipo de contenido que lleva. Para ello, tanto en expression como expression_comparacion vamos a indicar que son de tipo type, a través de un entero, que nos indicará el contenido que llevan. Para realizar una mejor identificación del tipo en tiempo de ejecución, nos ayudaremos de un array auxiliar, llamado tipos, que contendrá la siguiente información por posición:

0 -> “entero”
1 -> “float”
2 -> “boolean”
3 -> “error”

Por poner un pequeño ejemplo, cuando se realice

expression -> NUM

expression pasará a ser del tipo entero, por lo que el tipo asociado, representado con $$ pasará a ser 0, indicando que la expression es ya un entero. Si en vez de un entero fuese

expression -> DECIMAL

el tipo de la expression será 1, diferenciándose de los enteros por el valor 1.

¿Y si le asignamos un identificador? Aquí es donde vamos a hacer uso de una lista en la que, cuando un identificador se inicialice, se guardará la información para usos venideros. Por ello, cuando se asigne un identificador, se llamará a la función que nos da la lista para agregar un identificador, indicando si se trata de un entero o de un float. Por ello, en las reglas de producción:

```
asig            -> id = expression
asig_float      -> id = expression
```

Tendremos la siguiente instrucción, cambiando el tipo de cada asignación:

```
insertarVar(&lVar,$1,$3,0);
insertarVar(&lVar,$1,$3,1);
```

Y para conocer del tipo que es, se usa la siguiente función:

```
$$ = consultarVar(lVar,$1);
```

Ahora, toda expression tendrá un tipo asociado, pero, ¿qué ocurre cuando se realiza una expresión aritmética? Ya que la expression puede ser de tipo entero o de tipo float. Para ello, se comprobará el tipo de cada uno de ellos en la operación. Si ambos comparten el mismo tipo, el tipo de la expression será el mismo. En caso contrario, el tipo pasará a un valor 3, que nos indica que se ha producido un error.

Lo mismo ocurre para la expression_comparacion. Cuando se comparan dos expression, se tendrá en cuenta su tipo, y si no son el mismo, será un error. Al ser el mismo, el tipo pasará a ser 2, que corresponde que es un boolean.

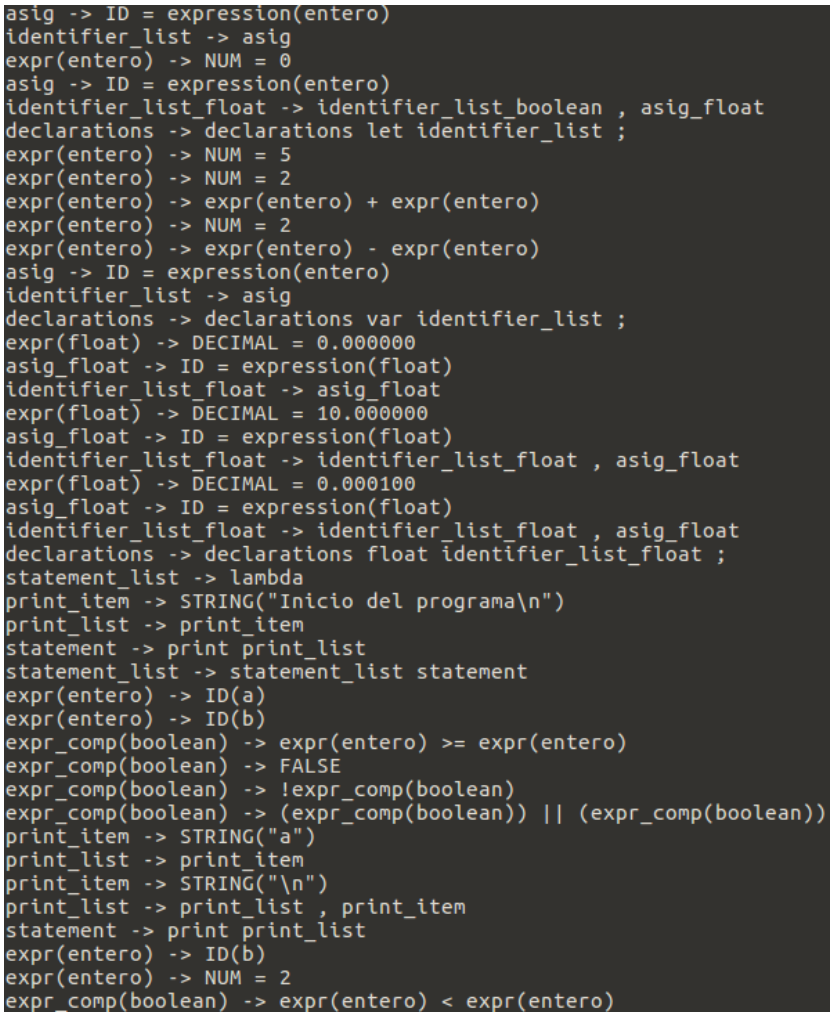
De esta manera, se resuelven los problemas de la gramática extendida.

FUNCIONAMIENTO

Para usar nuestra aplicación, usamos un Makefile que será el encargado tanto de generar los archivos necesarios para compilar nuestro programa (make), probar nuestro compilador (make run) y limpiar los archivos generados para compilar (make clean), de manera que solo deja los archivos fuentes.

La salida generada, gracias a los prints incluídos en el analizador sintáctico, nos permitirá las reglas que se van produciendo, de manera que podemos ver el camino que sigue el programa.

Pero dicha salida generada, para el archivo de prueba que se tiene preparado para el make es demasiado grande para una simple captura de pantalla. Por eso, se adjunta una captura de pantalla con una pequeña parte de la ejecución del programa de prueba:



```
asig -> ID = expression(entero)
identifier_list -> asig
expr(entero) -> NUM = 0
asig -> ID = expression(entero)
identifier_list_float -> identifier_list_boolean , asig_float
declarations -> declarations let identifier_list ;
expr(entero) -> NUM = 5
expr(entero) -> NUM = 2
expr(entero) -> expr(entero) + expr(entero)
expr(entero) -> NUM = 2
expr(entero) -> expr(entero) - expr(entero)
asig -> ID = expression(entero)
identifier_list -> asig
declarations -> declarations var identifier_list ;
expr(float) -> DECIMAL = 0.000000
asig_float -> ID = expression(float)
identifier_list_float -> asig_float
expr(float) -> DECIMAL = 10.000000
asig_float -> ID = expression(float)
identifier_list_float -> identifier_list_float , asig_float
expr(float) -> DECIMAL = 0.000100
asig_float -> ID = expression(float)
identifier_list_float -> identifier_list_float , asig_float
declarations -> declarations float identifier_list_float ;
statement_list -> lambda
print_item -> STRING("Inicio del programa\n")
print_list -> print_item
statement -> print print_list
statement_list -> statement_list statement
expr(entero) -> ID(a)
expr(entero) -> ID(b)
expr_comp(boolean) -> expr(entero) >= expr(entero)
expr_comp(boolean) -> FALSE
expr_comp(boolean) -> !expr_comp(boolean)
expr_comp(boolean) -> (expr_comp(boolean)) || (expr_comp(boolean))
print_item -> STRING("a")
print_list -> print_item
print_item -> STRING("\n")
print_list -> print_list , print_item
statement -> print print_list
expr(entero) -> ID(b)
expr(entero) -> NUM = 2
expr_comp(boolean) -> expr(entero) < expr(entero)
```