Práctica No. 5
Analizador sintáctico con programas en Lex y Yacc

Introducción

Podemos definir a Lex cómo una herramienta para construir analizadores léxicos o *lexers*. Un lexer lee un flujo de entrada (normalmente de `stdin` o de algun otro archivo) y lo divide en unidades léxicas (llamadas tokens) cómo un producto final o para ser procesadas por otro programa. Yacc (acrónimo de Yet Another Compiler Compiler) es una herramienta que, a partir de la descripción de una gramática que define un lenguaje, genera un analizador sintáctico (parser) para esa gramática. De manera muy simple, podemos decir que una gramática define o especifíca secuencias válidas de tokens válidos. Al igual que en Lex, un programa en Yacc se divide en 3 partes:

```
declaraciones
%%
reglas de producción
%%
código adicional
```

Declaraciones

La primera parte puede contener:

- Especificaciones escritas en el lenguaje destino, definidas entre `%{` y `%}` (cada símbolo colocado al principio de una línea)
- Declaraciones de tokens, con la palabra clave `%token`
- El tipo del terminal, con la palabra reservada `%union`
- Información sobre las prioridades de los operadores y su asociatividad.
- El axioma de la gramática, o símbolo inicial, usando la palabra reservada `%start` (si no se especifica, el axioma es la primera regla de la segunda parte del archivo).

La variable `yylval`, declarada implicitamente en el tipo `%union` es importante ya que contiene la descripción del último token leído.

Reglas de producción

Esta es la única parte necesaria, esto es, la que siempre debe aparecer en un programa yacc. Puede contener:

- Declaraciones y/o definiciones encerradas entre `%{` y `%}`.
- Reglas de producción de la gramática.

Las reglas de producción son definidas así:

```
expresion_no_terminal:
            cuerpo_1        { sentencias acciones 1 }
          | cuerpo_2        { sentencias acciones 2 }
          | ...
          :
          | cuerpo_n        { sentencias acciones n }
          ;
```

donde `cuerpo_i` pueden ser expresiones terminales o no terminales del lenguaje.

Código adicional

Esta parte contiene código adicional. En nuestro caso debe contener la función `main()` que debe llamar a la función `yyparse()`, y una función `yyerror(char *mensaje)`, que será llamada cuando ocurra un error de sintáxis.

Objetivo
El objetivo de esta práctica es combinar programas Lex y Yacc para desarrollar una calculadora sencilla.

Actividades
1.- Captura el siguiente programa lex y guardalo en un archivo llamado `calc.l`

```
/* calc.l */
%{
#define YYSTYPE double         /* redefine el tipo de yylval a flotante */
extern YYSTYPE yylval;
#include "calc_tab.h"          /* Archivo de cabecera es creado por yacc */
#include <stdlib.h>
%}

entero    [0-9]+
exponente [eE][+-]?{entero}
real      {entero}("."{entero})?{exponente}?

%%

[ \t]+              ; /* Se ignoran los caracteres en blanco */

{real}              {  yylval=atof(yytext); return(NUMERO);  }

[-+*/~()\n]     return(*yytext);

.               yyerror("caracter invalido\n");

%%

int yywrap(void)
{
    return 1;
}
```

2.- Captura el siguiente programa yacc y guardalo en un archivo llamado `calc.y`:

```
/* calc.y */
%{
#define YYSTYPE double         /* redefine el tipo de yylval */
extern YYSTYPE yylval;

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}
%token    NUMERO
%left     '+' '-'
%left     '*' '/'
%left     NEG
%right    '~'

%start prog
%%

prog:       /* Vacio */
          | prog expr '\n'     { printf("Resultado: %f\n",$2); }
          ;

expr:
            NUMERO             { $$=$1; }
          | expr '+' expr      { $$=$1+$3; }
```

```
          | expr '-' expr      { $$=$1-$3; }
          | expr '*' expr      { $$=$1*$3; }
          | expr '/' expr      { $$=$1/$3; }
          | '-' expr %prec NEG { $$=-$2; }
          | expr '~' expr      { $$=pow($1,$3); }
          | '(' expr ')'       { $$=$2; }
          ;
%%

int yyerror(char *msje) {
  printf("%s\n",msje);
}

int main(void) {
  yyparse();
  return 0;
}
```

3.- Compilación de los analizadores
El siguiente paso es compilar los programas yacc y lex. Primero debemos compilar el programa yacc y luego el programa lex.

3.1 Teclea el comando `bison -d calc.y<Enter>`
Que debe generar el código fuente del analizador sintáctico y el archivo de cabecera con las definiciones de los tokens. Usa el comando `dir calc_tab.*` para verificar los siguientes datos de los archivos generados:
Archivo: `calc_tab.c` Tamaño: _____
Fecha: _____
Archivo: `calc_tab.h` Tamaño: _____
Fecha: _____

3.2 Teclea el comando `flex -ocalc.c calc.l<Enter>`
Que debe generar el código fuente del analizador léxico. Usando el comando `dir calc.c` para verificar los siguientes datos de los archivos generados:
Archivo: `calc.c` Tamaño: _____
Fecha: _____

3.3 Generamos el ejecutable a partir del código fuente de los analizadores tecleando los siguiente comando:

```
tcc -ecalc.exe calc_tab.c calc.c<Enter>    (Si usas Turbo C)

lc calc_tab.c calc.c -o calc.exe<Enter>    (Si usas lcc-win)
```

Lo que debe generar el archivo `calc.exe`. Usando el comando `dir calc.exe` obtén:
Archivo: `calc.exe` Tamaño: _____
Fecha: _____

4.- Comprobación de la calculadora
Para ejecutar la calculadora, sólo tecleamos el nombre del ejecutable `calc<enter>`
4.1 Prueba las siguientes operaciones y reporta sus resultados:
`23+23` Resultado: _____
`14+(5-3)` Resultado: _____
`5+3*4` Resultado: _____
`3.1416*(23.2-4.1~2)` Resultado: _____

`(3+4*5)-(3/2+4)` Resultado: _____

Nota: Termina la ejecución de la calculadora con `Ctrl+D` o `CTRL+Z` o cualquier caracter diferente a los válidos para esta gramática.

5.- Modifica la calculadora para que también acepte la operación MODULO (x%y = X MODULO y)[1]

5.1.- ¿Qué debes agregar al archivo `calc.l`?

_____

_____

5.2.- ¿Qué debes agregar al archivo `calc.y`?

_____

_____

5.3.- Modifica los archivos, repite los pasos de compilación y prueba la nueva operación MODULO.

`25%5` Resultado: _____

`27%5` Resultado: _____

6.- Comentarios y conclusiones

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

1 Recuerda que la operación MODULO requiere operadores enteros.