Universidad Nacional Autónoma de México
Semestre 2020-2
Compiladores
Análisis Léxico
Alumnos: González Zepeda Felix
Mendoza Velázquez Daniel Adrián
Ramírez Ancona Simón Eduardo
Valeriano Barrios Cristian
Programa 1

a) Análisis del problema

i. Con base a la gramática presentada se requiere diseñar un analizador léxico que reconozca a los símbolos terminales, se utilizarán palabras similares a las de la gramática propuesta para detectar errores con mayor facilidad. Se tendrán palabras, números y signos representados como símbolos no terminales que el programa deberá poder retornar como TOKEN's esto lo hará identificando las expresiones regulares.

b) Diseño de la solución

Identificación de símbolos terminales para la gramática y su acción en análisis léxico.

i. Terminales	Acción en lexer.l
"estructura"	{ return ESTRUCTURA; }
"inicio"	{ return INICIO; }
"fin"	{ return FIN; }
"+"	{ return MAS; }
"-"	{ return MENOS; }
"*"	{ return MUL; }
"/"	{ return DIV; }
"%"	{ return MODULO; }
"{"	{ return A_LLAVE; }
"}"	{ return C_LLAVE; }
"("	{ return A_PAR; }
")"	{ return C_PAR; }
"["	{ return A_CORCHETE; }
"]"	{ return C_CORCHETE; }
";"	{ return PYC; }
"ent"	{ return ENT; }
"real"	{ return REAL; }
"dreal"	{ return DREAL; }
"car"	{ return CAR; }
"sin"	{ return SIN; }
">"	{ return S_MAYOR; }
"<"	{ return S_MENOR; }
">="	{ return S_MAYORIG; }
"<="	{ return S_MENORIG; }
"<>"	{ return DIFERENTE; }
"="	{ return IGUAL; }
","	{ return COMA; }

```
"def"                { return DEF;}
"."                  { return PUNTO; }
"si"                 { return SI; }
"entonces"           { return ENTONCES; }
"sino"               { return SINO; }
"mientras"           { return MIENTRAS; }
"hacer"              { return HACER; }
"segun"              { return SEGUN; }
"escribir"           { return ESCRIBIR; }
"leer"               { return LEER; }
"devolver"           { return DEVOLVER; }
":="                 { return ASIG; }
"terminar"           { return TERMINAR; }
"caso"               { return CASO; }
"pred"               { return PRED; }
":"                  { return D_PUNTOS; }
"o"                  { return OR; }
"y"                  { return AND; }
"no"                 { return NOT; }
"verdadero"          { return VERDADERO; }
"falso"              { return FALSO; }
{num}                { return NUM; }
{flotante}           { return NUM; }
{doble}              { return NUM; }
{caracter}           { return CARACTER; }
{cadena}             { return CADENA; }
{id}                 { return ID; }
```

ii. Producciones y gramáticas

Expresión regular	**Consideraciones y restricciones**
programa → declaraciones funciones	Sólo se considerará lo que las funciones tengan declarado
declaraciones → tipo lista_var ; declaraciones | tipo_registro lista_var ; declaraciones | ε	En declaraciones se admiten un tipo seguido de una lista de variables ó tipo de registro, lista de variables ó la cadena vacía. Se pueden concatenar varias declaraciones separadas por punto y coma.
tipo_registro → **estructura inicio** declaraciones fin	
tipo → base tipo_arreglo	**tipo** define el modelo de la variable a implementar
base → **ent** | **real** | **dreal** | **car** | **sin**	Define la base de nuestros tipos, pueden ser enteros, reales, dobles, caracteres o sin atributos.

tipo_arreglo → **[num]** tipo_arreglo | ε	**num** acepta números y a sí mismo para ampliar su tipo, o a una cadena vacía para finalizar.
lista_var → lista_var , **id** | **id**	Permite agregar más variables separadas con comas al asignar un tipo.
funciones → **def** tipo **id**(argumentos) **inicio** declaraciones sentencias **fin** funciones | ε	Permite declarar funciones que contengan declaraciones y sentencias dentro, se pueden declarar una o varias funciones.
argumentos → lista_arg | **sin**	**argumentos** acepta a 'lista argumentos', ó puede ser una cadena vacía
lista_arg → lista_arg, arg | arg	**lista argumentos** se acepta a sí misma seguida de 'tipo', 'id', 'parte', 'arreglo'; ó puede aceptar lo anteriormente mencionado sin considerarse a sí misma.
arg → tipo_arg **id**	**parte arreglo** está compuesto por un tipo de arreglo, seguido de un identificador.
tipo_arg → base param_arr	**tipo_arg** está compuesto por la base del tipo de dato, seguido de los parámetros del arreglo.
param_arr → **[ ]** param_arr | ε	**param_arr** permite declara arreglos pero empezando por el tamaño, pueden ser varios tamaños.
sentencias → sentencias sentencia | sentencia	**sentencias** Permite declarar una o varias sentencias.
sentencia → **si** e_bool **entonces** sentencia fin | **si** e_bool entonces sentencias **sino** sentencia **fin** | **mientras** e_bool **hacer** sentencia **fin** | **hacer** sentencia **mientras** e_bool; | **segun** (variable) **hacer** casos predeterminados **fin** | variable := expresion ; | **escribir** expresion ; | **leer** variable ; | **devolver;** | **devolver** expresion; | **terminar;** | **inicio** sentencias **fin**	**sentencia** se puede aceptar a sí misma, ó pueden ser condicionales ('if', 'while', 'do', 'for') acompañados de condición y de sí misma. Puede ser un 'return', 'switch', 'break', 'print' acompañado de sí misma.
casos → **caso num:** sentencia casos | **caso num:** sentencia	**casos** requiere de 'numero' y de 'sentencia' para ser implementado seguido de un 'predeterminado' , ó puede ser una cadena vacía

predeterminado → **pred:** sentencia | ε	**predeterminado** acepta a 'default:', seguido de una sentencia; ó puede ser una cadena vacía
e_bool → e_bool **o** e_bool | e_bool **y** e_bool | **no** e_bool | relacional | **verdadero** | **falso**	**e_bool** permite declarar expresiones booleanas o directamente verdadero o falso.
relacional → relacional > relacional | relacional < relacional | relacional <= relacional | relacional >= relacional | relacional <> relacional | relacional = relacional | expresion	**relacional** admite comparaciones de tipo mayor que (>), menor que (<), menor o igual que (<=), mayor o igual que (>=), diferente de (<>), igual que (=), además también admite sustitución por una expresión.
expresion→expresion+expresion |expresion−expresion |expresion∗expresion |expresion/expresion |expresion%expresion |**(**expresion**)** |variable|**num**|**cadena**|**caracter**	**expresión** acepta suma (+), resta (-), multiplicación(*), división(/), módulo (%) hacia sí mismo; al igual que acepta agrupar más expresiones mediante paréntesis, admite variables, números, cadenas y caractéres.
variable → id variable_comp	**variable** permite declarar el nombre de una variable y sus opciones complementarias para arreglos, estructuras y parámetros.
variable_comp → dato_est_sim | arreglo | (parametros)	**variable_comp** puede ser un arreglo, estructura o parámetro perteneciente a una variable.
dato_est_sim → dato_est_sim .id | ε	**dato_est_sim** permite acceder a una variable interna de una estructura o a varias.
arreglo → [expresion] | arreglo [expresion]	**arreglo** puede contener las operaciones básicas entre dos expresiones y también las puede agrupar o hacer referencia una variable o a un número, cadena o caracter.
parametros → lista param | ε	**parametros** puede ser 'lista param' o una cadena vacía
lista_param → lista_param, expresion | expresion	**lista param** se acepta a sí misma seguido de 'expresion' o solo acepta a 'expresion'

iii. AFD

c) Implementación

i. Primero se identificó los símbolos terminales de la gramática para poder implementarlos en el programa lexer.l esto con el fin de que el analizador léxico propuesto pudiera identificar dichos terminales por medio de tokens.

ii. Posterior a la identificación de los símbolos no terminales se realizó un análisis de la gramática identificando sus partes y una breve descripción de su función.

iii. Para el lexer.l que es en donde definimos los terminales que serán identificados por flex.
El contador de lineas nos sirve para identificar en qué parte de la cadena prueba está cada uno de los elementos que reconoce el analizador léxico.

```
/*Indicar que solo lea un fichero de entrada*/
%option noyywrap
/*contador de lineas*/
%option yylineno
```

Las expresiones regulares que utilizamos son las siguientes:

```
/*Identificadores*/

letra [a-zA-Z]
caracter (\'{letra}\')
digito [0-9]
digitos {digito}+
num  (([1-9]{digito}*)|[0])
exp [eE][+-]?{digitos}
doble (("."{digitos}|{digitos}"."{digitos}){exp}?|{digitos}{exp})
flotante (("."{digitos}|{digitos}"."{digitos}){exp}?|{digitos}{exp})[fF]
id  ({letra}|_)({letra}|{digito}|_)*
cadena \"[^"]*\"
```

En esta sección declaramos los terminales que serán identificados por flex. Solo se usan algunos ejemplos, el código queda en manos del usuario si desea ver los terminales completos.

```
%%
"estructura"        { return ESTRUCTURA; }
"inicio"            { return INICIO; }
"fin"               { return FIN; }
"+"                 { return MAS; }
"-"                 { return MENOS; }
"*"                 { return MUL; }
"/"                 { return DIV; }
"%"                 { return MODULO; }
"{"                 { return A_LLAVE; }
"}"                 { return C_LLAVE; }
"("                 { return A_PAR; }
")"                 { return C_PAR; }
"["                 { return A_CORCHETE; }
"]"                 { return C_CORCHETE; }
```

```
";"                 { return PYC; }
"ent"               { return ENT; }
"real"              { return REAL; }
"dreal"             { return DREAL; }
"car"               { return CAR; }
"sin"               { return SIN; }
{num}               { return NUM; }
{id}                { return ID; } /*En caso de encontrar algo del
identificador*/
[ \t\n\r]+          {}          /*Para que espacios en blanco sean
ignorados*/
.                   {  error(yytext); } /*Para los simbolos no
reconocidos error lexico*/
%%
```

iv. Para la definición de terminales se utilizó un archivo llamado tokens.h para hacer pruebas antes de utilizar el archivo parser.y de los cuales solo se incluyen algunos ejemplos.

```
#ifndef TOKENS_H
#define TOKENS_H
```

Se les asigna una dirección de memoria a cada uno de los terminales.

```
#define ESTRUCTURA 200
#define INICIO 201
#define FIN 203
#define MAS 204
#define MENOS 205
#define MUL 206
#define DIV 207
#define MODULO 208
#define A_LLAVE 209
#define C_LLAVE 210
#define A_PAR 211
#define C_PAR 212
#define A_CORCHETE 213
#define C_CORCHETE 214
#define PYC 215
#define ENT 216
#define REAL 217
#define DREAL 218
#define CAR 219
#define SIN 220
#endif
```

d) Ejecución del programa

Para la ejecución del programa se escribió un script de shell (Linux) llamado ***tester.sh*** que compila el lexer, el parser y el main ejecutando la prueba un un archivo llamado prueba. Se anexa el código:

```
#! /bin/bash
flex lexer.l
```

```
byacc -d parser.y
gcc lex.yy.c y.tab.c main.c -o main
./main prueba
```

e) Función principal

Para la función principal se desarrolló un archivo *main.c* que lee un archivo desde terminal como argumento, detecta que exista el archivo que lo pueda abrir y leer en conjunto con el archivo *lexer.l* para retornar los tokens y el *parser.y* para comprobar que el programa del archivo prueba esté correcto según la gramática, finalmente al encontrar el fin de archivo cierra el archivo prueba. Se anexa código:

```
int main(int argc, char** argv){
   FILE *f;
   if(argc < 2){
       printf("Faltan argumentos\n");
       return -1;
   }
   f = fopen(argv[1],"r");
   if(!f){
       printf("El archivo %s no se puede abrir\n",argv[1]);
       return -1;
   }
   yyin = f;
   yyparse();
   fclose(yyin);
}
```

f) Documentación de código

g) Modificaciones

i. lexer.l

```
/* Creación Felix G. 23/05/20 */
/* Definición de símbolos no terminales, uso de bibliotecas*/
/*y variables especiales de flex*/
```

```
/*Expresiones regulares*/
/* Modificación Simón R. 24/05/20*/
/*Se separan los símbolos terminales para que retornen token
individuales.*/
```

ii. tokens.h

```
/* Creación Felix G. 23/05/20
  Definición de tokens y variables especiales de flex
  DEFINICION DE LOS TERMINALES
  Correccion de definicion de terminales Daniel V. 23/05/20
*/
```

```
/* Modif. Cristian Valeriano 24/05/2020 se agrego tokens faltantes
```

```
  Modif. Felix Gonzalez 25/05/2020 se agrego numero de direccion de
memoria
  Definimos las cosntantes para cada simbolo terminal
*/
```