Compiladores
Análisis Sintáctico
Programa 2

A. **Descripción del problema.** Se presenta la siguiente gramática para la cual se requiere hacer un análisis sintáctico, donde en caso de que exista ambigüedad se debe quitar, así como la recursividad o en otro posible caso de factorizar. Una vez terminado dicho análisis se deberá programar el analizador sintáctico utilizando bison o yacc.

sin: significa sin tipo, car: tipo caracter

1. programa → declaraciones funciones
2. declaraciones → tipo_lista_var; declaraciones | tipo_registro lista_var; declaraciones | ε
3. tipo_registro → **estructura inicio** declaraciones **fin**
4. tipo → base tipo_arreglo
5. base → **ent** | **real** | **dreal** | **car** | **sin**
6. tipo_arreglo → **(num)** tipo_arreglo | ε
7. lista_var → lista_var, **id** | **id**
8. funciones → **def** tipo **id**(argumentos) **inicio** declaraciones sentencias fin funciones | ε
9. argumentos → listar_arg | **sin**
10. lista_arg → lista_arg, arg | arg
11. arg → tipo_arg **id**
12. tipo_arg → base param_arr
13. param_arr → **()** param_arr | ε
14. sentencias → sentencias sentencia | sentencia
15. sentencia → **si** e_bool **entonces** sentencia **fin** | **si** e_bool **entonces** sentencia **sino** sentencia **fin** | **mientras** e_bool **hacer** sentencia **fin** | **hacer** sentencia **mientras** e_bool;
| **segun** (variable) **hacer** casos predeterminado **fin** | variable := expresion ; | **escribir** expresion ; | **leer** variable ; | **devolver**; | **devolver** expresion; | **terminar**; | **inicio** sentencias **fin**
16. casos → **caso num**: sentencia casos | **caso num**: sentencia
17. predeterminado → **pred**: sentencia | ε
18. e_bool → e_bool **o** e_bool | e_bool **y** e_bool | **no** e_bool | (e_bool) | relacional
| **verdadero** | **falso**
19. relacional → relacional oprel relacional | expresion
20. oprel → > | < | >= | <= | <> | =
21. expresion → expresion oparit expresion | expresion % expresion | (expresion) | id
| variable | **num** | **cadena** | **caracter** | **id**(parametros)
22. oparit → + | _ | . | /
23. variable → dato_est_sim | arreglo
24. dato_est_sim → dato_est_sim .**id** | **id**
25. arreglo → **id** (expresion) | arreglo (expresion)
26. parametros → lista_param | ε
27. lista_param → lista_param, expresion | expresion

B. Diseño de la solución

Para diseñar la solución a este problema, es necesario tener conocimientos de las materias de lenguajes formales y autómatas, así como los temas previos a este en compiladores. Inicialmente se analizó la gramática dada, y con el programa 1 que se desarrolló previamente, se determinaron los símbolos Terminales y No Terminales. Asi como se llego a la creación del analizador léxico utilizando flex. Dicho analizador léxico lo usaremos junto con el analizador sintáctico al que lleguemos en esta tarea.

1. Incluir diagramas de sintaxis

2. En caso de quitar ambigüedad incluir el proceso

3. En caso de eliminar recursividad incluir el proceso

4. En caso de factorizar incluir el proceso.

C. Implementación

Nuestro código se dividió en 3 secciones, la sección de declaraciones,

```
%{
  #include <stdio.h>
    #include <string.h>
    void yyerror(char *s);
%}
```

En el siguiente bloque de código se indican los símbolos Terminales.

```
%token ESTRUCT INICIO END ENTERO REAL DREAL ...
%token ID CADENA CARACTER
%token PYC DOSP
```

Aquí también estamos indicando símbolos Terminales pero con

```
%left MAS MENOS
%left MUL DIV
%left MODU
```

Para la declaración de los No Terminales que tienen un tipo, se utilizó el siguiente código.

```
%type<eTipo> tipo base
%type<eTipo> tipo_arreglo
%type<eExpresion> expresion
```

En la unión se asigna un atributo de tipo “float (flotante)” con el nombre de “real” y se inician las reglas de producción con el símbolo inicial llamado “programa”.

```
%union
{
  float real;
}
%start programa
%%
```

A continuación, se declaran las reglas de producción de nuestra gramática previamente proporcionada.

```
programa        : declaraciones funciones {}

declaraciones   : tipo {type = $1} lista_var PYC declaraciones {}
                | tipo_registro lista_var PYC declaraciones {}
                | /* epsilon */
```

Al final se declara este pequeño bloque, pues es lo que necesitaremos para todo el programa.

```
void yyerror(char* s);
extern int yylex();
extern int n;
```

NOTA: Los bloques de código mostrados no son los bloques completos, esto por temas de simplicidad.

D. Forma de ejecutar el programa

Windows: Desde “CMD” debemos llegar a la locación del archivo, posteriormente teclear la siguiente instrucción:

```
C:\Users\      \        \  \Compiladores\Programas_Equipo>bison -dy "nombre_del_programa".y
```

Después con la siguiente instrucción enlazamos los archivos con un archivo main.c (Posteriormente en el curso lo veremos y haremos a detalle):

```
C:\Users\      \        \  \Compiladores\Programas_Equipo>gcc y.tab.c lex.yy.c main.c -0 "nombre del archivo".exe
```

El “nombre del archivo” es el archivo que se generó de la primera instrucción.

Linux: Desde la terminal de linux, nos posicionamos en el directorio donde se encuentra nuestro programa y ejecutamos el siguiente comando:

```
osmar@pc-oja ~/Desktop/Compiladores/2020-2Programas_usr1/Programa2 $ bison -d "archivo_del_programa".y
```

En caso de no saber para qué sirve la bandera -d, puede consultar mediante la línea de comandos la lista de los diferentes modos en los que se puede ejecutar un archivo de bsion con una breve explicación de para que se usa cada uno. Esto se hace con el siguiente comando:

```
osmar@pc-oja ~/Desktop/Compiladores/2020-2Programas_usr1/Programa2 $ bison --help
Usage: bison [OPTION]... FILE
Generate a deterministic LR or generalized LR (GLR) parser employing
LALR(1), IELR(1), or canonical LR(1) parser tables.  IELR(1) and
canonical LR(1) support is experimental.

Mandatory arguments to long options are mandatory for short options too.
The same is true for optional arguments.

Operation modes:
  -h, --help                 display this help and exit
  -V, --version              output version information and exit
      --print-localedir      output directory containing locale-dependent data
      --print-datadir        output directory containing skeletons and XSLT
  -y, --yacc                 emulate POSIX Yacc
  -W, --warnings[=CATEGORY]  report the warnings falling in CATEGORY
  -f, --feature[=FEATURE]    activate miscellaneous features

Parser:
  -L, --language=LANGUAGE          specify the output programming language
  -S, --skeleton=FILE              specify the skeleton to use
  -t, --debug                      instrument the parser for tracing
                                   same as '-Dparse.trace'
      --locations                  enable location support
  -D, --define=NAME[=VALUE]        similar to '%define NAME "VALUE"'
  -F, --force-define=NAME[=VALUE]  override '%define NAME "VALUE"'
  -p, --name-prefix=PREFIX         prepend PREFIX to the external symbols
                                   deprecated by '-Dapi.prefix=PREFIX'
  -l, --no-lines                   don't generate '#line' directives
  -k, --token-table                include a table of token names
```

El siguiente paso es ejecutar todos los archivos .c que genere bson así como los archivos .c generados por el analizador léxico que creamos en el programa 1 utilizando flex. Asi como tambien nuestro archivo main que hemos creado.

```
osmar@pc-oja ~/Desktop/Compiladores/2020-2Programas_usr1/Programa2 $ gcc lex.yy.c y.tab.c main.c -o programa2
```

La bandera -o nos servirá para indicar cómo queremos que se llame nuestro programa ejecutable. Por lo que lo que viene después de esta bandera queda a consideración de cada persona.