UNIVERSIDAD NACIONAL DE SAN AGUSTÍN

FACULTAD DE PRODUCCION Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

LABORATORIO - TEORIA DE LA COMPUTACIÓN

Instalación y Reconocimiento del entorno Bison

Apaza Calsin Michael Secarlos

CUI: 20180564

Arequipa - Perú

2021

I OBJETIVOS

- Instalar el software Bison
- Utilizar las funcionalidades básicas de Bison
- Crear una aplicación básica en Bison

II EJERCICIOS PROPUESTOS

La resolución de ejercicios puede hacerse aceptando la expresión una sola vez o también de manera iterativa.

1. Crear un analizador sintáctico que reconozca cinco combinaciones de notas musicales. (2 puntos)
2. Crear un analizador sintáctico que reconozca la apertura de paréntesis y el cierre de paréntesis y que en medio de ellos pueda aceptar texto y números. (3 puntos)
3. Crear un analizador sintáctico que permita la asignación de valor entero a una variable. Por ejemplo se debe aceptar: (3 puntos)
   num001=35
   v_0=13
   numero=76
   Aclaración, el nombre de la variable tiene que empezar con letra pero puede tener numero y guion bajo.
4. Crear un analizador sintáctico que permita la declaración de variables tipo int, double y string, que terminen en ';'. Por ejemplo se debe aceptar: (3 puntos)
   int num001;
   double value_0;
   string dia;
5. Crear un analizador sintáctico que permita reconocer la condicional de un variable con un numero decimal. Por ejemplo: (4 puntos)
   if (cantidad == 2.3)
   if (cant002 >= 1.73)
   if (cantidad < 9.3)
6. Modificar el analizador sintáctico de la calculadora para que pueda recibir la operación modulo y potencia. (4 puntos)

III RESOLUCIÓN

Para estos ejercicios, se usaron dos archivos, el primero **ejercicio#.l**, que representa el archivo que se compila con flex, luego el archivo **sintactico#.y** que es el archivo para bison, los símbolos #, expresan el número del ejercicio propuesto.

para todos los ejercicios se usó la estructura básica tanto para flex como para Bison vista en clases de laboratorio.

1. **Ejercicio 1**
   Para este ejercicio se definió el token NOTAS, usado de la siguiente forma:

NOTAS ((do|DO|re|RE|mi|MI|fa|FA|so|SO|la|LA|si|SI)[ ]*){5}

De forma que se aceptan cadenas de las notas en minúsculas y/o mayúsculas, seguidas de 0 o más espacios de separación, todo esto solamente veces:

```
%{
    #include "sintactico1.tab.h"
    int yyparse();
%}
NOTAS ((do|DO|re|RE|mi|MI|fa|FA|so|SO|la|LA|si|SI)[ ]*){5}
%%
{NOTAS}           {return NOTAS;}
\n return *yytext;
%%

int yywrap(){
    yyparse();
    return 0;
}
```

Luego en el archivo sintáctico de bison(**.y**), se define el token NOTAS, y la cadena que acepta es solamente este TOKEN, además de estar definido de forma iterativa.

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

%token NOTAS
%%
cadena: NOTAS '\n' {printf("5 notas seguidas\n");};
| cadena NOTAS '\n' {printf("5 notas seguidas\n");};
%%

int yyerror (char *s)
{
    printf ("%s\n", s);
    return 0;
}
int main()
{
    yyparse();
    return 0;
}
```

2. **Ejercicio 2**

Para este ejercicio se definieron 3 nombres para los tres tipos de datos que requiere el problema en el archivo **ejercicio2.l**: paréntesis de apertura, texto y paréntesis de cierre, y se definieron de la siguiente forma:

APERTURA **\(**
CIERRE **\)**
TEXTO **[a-zA-Z0-9]***

Los símbolos “(“ y “)” se expresan con contra slash, mientra que el texto puede aceptar cualquier letra minúscula, mayúsculas y números.
Esto de la siguiente forma:

```
%{
    #include "sintactico2.tab.h"
    int yyparse();
%}
APERTURA    \(
CIERRE      \)
TEXTO       [a-zA-Z0-9]*
%%
{APERTURA}          {return APERTURA;}
{CIERRE}           {return CIERRE;}
{TEXTO}           {return TEXTO;}
\n return *yytext;
%%

int yywrap(){
    yyparse();
    return 0;
}
```

Luego, en el archivo **sintactico2.y**, recibimos estos 3 tokens mencionados anteriormente, de forma que la cadena aceptada esté conformada en el siguiente forma:
APERTURA TEXTO CIERRE
Esta implementación también está hecha de forma iterativa:

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

%token APERTURA
%token TEXTO
%token CIERRE
%%
cadena: APERTURA TEXTO CIERRE '\n' {printf("Texto entre parentesis\n");};
| cadena APERTURA TEXTO CIERRE '\n' {printf("Texto entre parentesis\n");};
%%

int yyerror (char *s)
{
    printf ("%s\n", s);
    return 0;
}
int main()
{
    yyparse();
    return 0;
}
```

3. **Ejercicio 3**

En el archivo flex, mencionamos definimos las tres estructuras o partes necesarias para cada una de las cadenas que se tengan como entrada: el tipo de dato, el nombre de la variable y el símbolo “**;**”, que indica el final de la expresión:

VARIABLE **[a-zA-Z][a-zA-Z0-9_]***
ASIGNACION **=**
ENTERO **[0-9]+**

La variable como mínimo cuenta con una letra seguida de cualquier caracter o número e incluso el subguión.
La asignación solamente tiene el símbolo =.
El entero como mínimo solo contiene un número.
Viéndose de la siguiente forma en el archivo .l:

```
%{
    #include "sintactico3.tab.h"
    int yyparse();
%}
VARIABLE     [a-zA-Z][a-zA-Z0-9_]*
ASIGNACION   =
ENTERO       [0-9]+
%%
{VARIABLE}      {return VARIABLE;}
{ASIGNACION}    {return ASIGNACION;}
{ENTERO}        {return ENTERO;}
\n return *yytext;
%%

int yywrap(){
    yyparse();
    return 0;
}
```

Y el archivo .y contiene únicamente la concatenación de estos tokens:

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

%token VARIABLE
%token ASIGNACION
%token ENTERO
%%
cadena: VARIABLE ASIGNACION ENTERO '\n' {printf("Asignacion de un entero\n");};
| cadena VARIABLE ASIGNACION ENTERO '\n' {printf("Asignacion de un entero\n");};
%%

int yyerror (char *s)
{
    printf ("%s\n", s);
    return 0;
}
int main()
{
    yyparse();
    return 0;
}
```

4. **Ejercicio 4**

De la misma forma que el ejercicio anterior, se usa la misma expresion para las variables, luego se añaden el tipo de dato y el simbolo terminal ";".

TIPO **(int|double|string)**
VARIABLE **[a-zA-Z][a-zA-Z0-9_]***
TERMINAL **[;]**

El tipo de dato, sólo se expresa una vez, y puede ser de tipo entero, double o string luego la misma expresión de la variable vista antes, y finalmente el símbolo ";".

```
%{
    #include "sintactico4.tab.h"
    int yyparse();
%}
TIPO        (int|double|string)
VARIABLE    [a-zA-Z][a-zA-Z0-9_]*
TERMINAL    [;]
%%
{TIPO}          {return TIPO;}
{VARIABLE}      {return VARIABLE;}
{TERMINAL}      {return TERMINAL;}
\n return *yytext;
%%

int yywrap(){
    yyparse();
    return 0;
}
```

Y de la misma forma, se expresaron los tokens en el archivo bison: **sintactico4.y**

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

%token TIPO
%token VARIABLE
%token TERMINAL
%%
cadena: TIPO VARIABLE TERMINAL'\n' {printf("Declaracion de una variable\n");};
| cadena TIPO VARIABLE TERMINAL'\n' {printf("Declaracion de una variable\n");};
%%

int yyerror (char *s)
{
    printf ("%s\n", s);
    return 0;
}
int main()
{
    yyparse();
    return 0;
}
```

5. **Ejercicio 5**

Para este ejercicio, se hizo uso de las reglas ya definidas anteriormente: VARIABLE, APERTURA y CIERRE, y se añadieron 2 más, la primera que indica el nombre de la estructura IF, y la segunda que es un operador de comparación:

```
IF          if
APERTURA    \(
VARIABLE    [a-zA-Z][a-zA-Z0-9_]*
OPERADOR    (==|<=|>=|<|>)
CIERRE      \)
```

La regla de operador, solo selecciona los operadores lógicos más usados, como lo son los comparadores de números. El archivo **ejercicio5.l** se ve de la siguiente forma:

```
%{
    #include "sintactico5.tab.h"
    int yyparse();
%}
IF          if
APERTURA    \(
VARIABLE    [a-zA-Z][a-zA-Z0-9_]*
OPERADOR    (==|<=|>=|<|>)
CIERRE      \)
%%
{IF}            {return IF;}
{APERTURA}      {return APERTURA;}
{VARIABLE}      {return VARIABLE;}
{OPERADOR}      {return OPERADOR;}
{CIERRE}        {return CIERRE;}
\n return *yytext;
%%

int yywrap(){
    yyparse();
    return 0;
}
```

Mientras que para el archivo **sintactico5.y** tenemos:

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

%token IF
%token APERTURA
%token VARIABLE
%token OPERADOR
%token CIERRE
%%
cadena: IF APERTURA VARIABLE OPERADOR VARIABLE CIERRE'\n' {printf("Comparacion con if\n");};
| cadena IF APERTURA VARIABLE OPERADOR VARIABLE CIERRE'\n' {printf("Comparacion con if\n");};
%%

int yyerror (char *s)
{
    printf ("%s\n", s);
    return 0;
}
int main()
{
    yyparse();
    return 0;
}
```

6. Ejercicio 6

Para este ejercicio, se entiende que las operaciones de exponenciacion y modulo, se comportan o tienen la misma prioridad que la multiplicacion o la division, entonces se

consideró como **factor** en vez de **expresion**, entonces en el archivo **ejercicio6.l**, definimos el operador que nos brindará estas características:

```
"^"     { return EXP; }
"%"     { return MOD; }
```

Únicamente esto en el archivo **.l**, y para el archivo **sintactico6.y**, se definió las reglas, primero incluyendo la librería <math.h> en la cabecera de este archivo para realizar la operación de exponenciación.

```
%{
    #include <stdio.h>
    #include <math.h>
    int yylex();
    int yyerror (char *s);
%}
```

Luego en la sección de factor, se le considera a este de la misma forma, usando la función **pow**, para elevar un numero y **%** para obtener el modulo de dicho número con respecto de otro.

```
factor: term
    | factor MUL term { $$ = $1 * $3; ;printf("= %d ",$$);}
    | factor DIV term { $$ = $1 / $3; ;printf("= %d ",$$);}
    | factor EXP term { $$=pow($1,$3); ;printf("= %d ",$$);}
    | factor MOD term { $$ = $1 % $3; ;printf("= %d ",$$);};
```

Nota: investigando encontre que para que no existan errores a la hora de compilar, es mejor usar la opción -lm para que todo funcione correctamente, de la siguiente forma

```
gcc lex.yy.c sintactico6.tab.c -o salida6.out -lm
```

IV CUESTIONARIO

1. En el código de la calculadora ¿Porque se define exp y factor como símbolos no terminales y por qué están separadas las operaciones de ADD y SUB con respecto a MUL DIV? (1 punto)

```
14 exp: factor
15  | exp ADD factor { $$ = $1 + $3; ;printf("= %d ", $$);}
16  | exp SUB factor { $$ = $1 - $3; ;printf("= %d ", $$);} ;
17
18 factor: term
19  | factor MUL term { $$ = $1 * $3; ;printf("= %d ", $$);}
20  | factor DIV term { $$ = $1 / $3; ;printf("= %d ", $$);} ;
21
22 term: NUMBER;
23 %%
```

Porque en ecuaciones normales, se da prioridad a las operaciones de multiplicación y división sobre las de suma o resta, entonces siguiendo esta regla, en este caso el orden sería: empezar desde **term**, luego los **factor**, luego las **exp**, y finalmente la **calclist**.
Donde **term** son números, **factor** son numeros operador por multiplicacion o division, **exp**, son las sumas y restas, y finalmente **calclist** almacena todos estos números y operaciones.