UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Trabajo Práctico N° 3 FLEX Y BISON

MATERIA: Sintaxis Y Semántica de los Lenguajes.

PROFESORA: Roxana María Leituz.

COMISIÓN: K2055.

ALUMNOS: Aylén Marta Chazarreta, Brisa Nelly Calzado, Dante Silva, Enrique Antonio Marques y Lisandro Corrales Cespedes.

FECHA DE ENTREGA: 24/11/2024.

Código en el archivo de flex (.l)

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include "y.tab.h"
%}
DIGITO [0-9]

LETRA [a-zA-Z]

IDENTIFICADOR {LETRA}({LETRA}|{DIGITO})*

constEntera {DIGITO}({DIGITO})*
%%
"inicio" {return INICIO;}
"fin" {return FIN;}
"escribir" {return ESCRIBIR;}
"leer" {return LEER;}


":=" {return ASIGNACION;}
{constEntera} {yylval.num=atoi(yytext);return CONSTANTE;}
{IDENTIFICADOR} {return ID;}
";" {return PYCOMA;}
"," {return COMA;}
"(" {return PARENIZQUIERDO;}
")" {return PARENDERECHO;}
"+" {return SUMA;}
"-" {return RESTA;}
%%
```

En primer lugar, partimos del archivo flex base compartido en clase y lo completamos con palabras reservadas tales como: inicio, fin, escribir y leer, las cuales devuelven los tokens correspondientes.

Código en el archivo de bison (.y)

En el código de bison empezamos con las declaraciones en C.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define LONGITUD_IDS 32
#define MAX_TABLA 200
#define LONGITUD_VECTOR 25

extern char *yytext;
extern int yyleng;
extern int yylex(void);
extern void yyerror(char*);

void verificarLongId(int);

typedef struct {
    char id[LONGITUD_IDS + 1];  // 32 caracteres para el id
    int valor;
} SIMBOLO;
static SIMBOLO vectorDeVariables[LONGITUD_VECTOR];
static SIMBOLO tablaSimbolos[MAX_TABLA];

int cardinalTabla = 0;
int cardinalVector = 0;
```

Declaramos las bibliotecas usadas, al igual que las macros. Asimismo declaramos variables externas. También definimos estructuras y contadores.

```
void iniciarTabla();
void iniciarVectorSimbolos();
int insertarSimbolo(char*);
int modificarSimbolo(char* , int);
int buscarSimbolo(char* );
int insertarSimboloVector(char * );
int buscarSimboloVector(char* ) ;
%}
```

Acá aparecen todos los prototipos de las funciones usadas más adelante.

```
%union {
    char* cadena;
    int num;
}

%token ASIGNACION PYCOMA COMA SUMA RESTA PARENIZQUIERDO PARENDERECHO INICIO FIN LEER ESCRIBIR
%token <cadena> ID
%token <num> CONSTANTE

%%
```

Con el %union declaramos un conjunto de datos, de tipo char* e int, que pueden ser utilizados para almacenar los valores asociados con los tokens. Por ejemplo, el token ID es una cadena y el token CONSTANTE es un número. Asimismo definimos tokens como ASIGNACIÓN, COMA, etc que son los tokens que el analizador sintáctico reconocerá.

Luego empezamos con las reglas gramaticales:

```
programa:
    { printf("Ingrese codigo en lenguaje micro\n"); }
    INICIO sentencias FIN
    { printf("Programa terminado con exito"); return 0; }
;

sentencias:
    sentencias sentencia
    | sentencia
;
```

- Un programa debe comenzar con la palabra clave inicio y terminar con fin.
- Entre inicio y fin se pueden incluir múltiples sentencias.
- Se imprimen mensajes al inicio y final.

```
sentencia:
    ID { verificarLongId(yyleng); }
    ASIGNACION expresion PYCOMA
    | LEER PARENIZQUIERDO listaVariables PARENDERECHO PYCOMA
    | ESCRIBIR PARENIZQUIERDO parametros PARENDERECHO PYCOMA
;
```

- Una sentencia puede ser una asignación, una lectura para ingresar valores o una escritura para mostrar resultados

```
expresion:
    primaria
    | expresion operadorAditivo primaria
;
```

- Una expresión puede ser una sola (primaria) o Una combinación de dos expresiones unidas

```
listaVariables:
    listaVariables COMA ID { verificarLongId(yyleng); }
    | ID { verificarLongId(yyleng); }
;
```

- Una lista de variables se define como una o más variables separadas por comas

```
parametros:
    parametros COMA expresion
    | expresion
;
```

- Esta regla sirve para definir lo que puede ir dentro del paréntesis de la instrucción escribir().

```
primaria:
    ID { verificarLongId(yyleng); if(buscarSimbolo(yytext)==-1){
      yyerror("la variable no fue declarada previamente");
    } }
    | CONSTANTE { printf("valores %d", atoi(yytext)); }
    | PARENIZQUIERDO expresion PARENDERECHO
;
```

- Definimos lo que puede ir en una operación o expresión, que puede ser una constante, un ID, paréntesis.

```
operadorAditivo:
    SUMA
    | RESTA
;

%%
```

- Reconoce los operadores básicos de suma y resta

Y ahí terminaron las reglas gramaticales

Y por último hicimos el código C adicional.

```
int main() {
    iniciarTabla();
    iniciarVectorSimbolos();
    yyparse();
}

void yyerror(char *s) {
    printf("Error %s\n", s);
}
```

Función que hicimos para mostrar mensajes de error personalizados si algo sale mal durante el análisis

```
void verificarLongId(int yyleng) {
    if (yyleng > LONGITUD_IDS)
        yyerror("Lexico: el ID supera la longitud maxima");
}
```

Esta función lo que hace es verificar la longitud del identificador ya que micro trabaja hasta con longitud 32

```
void iniciarTabla() {
    int i;
    for (i = 0; i < MAX_TABLA; i++) {
        tablaSimbolos[i].id[0] = '\0';  // Inicializa las cadenas a vacío
        tablaSimbolos[i].valor = 0;
    }
}
```

Esta función inicializa la tabla de símbolos

```
void iniciarVectorSimbolos(){
  int i;
  for(i = 0; i < LONGITUD_VECTOR; i++) {
    vectorDeVariables[i].id[0] = '\0';
    vectorDeVariables[i].valor = 0;

}}
```

Función que sirve para inicializar el vector de variables

```
int buscarSimbolo(char* nombre) {
    for (int i = 0; i <cardinalTabla; i++) {
        if (strcmp(tablaSimbolos[i].id, nombre) == 0) {
            return i;  // Retorna el índice del símbolo encontrado
        }
    }
    return -1;  // No encontrado
}
```

esta función lo que hace es buscar el identificador en la tabla de símbolos , si está retorna su posición y en el caso contrario retorna -1

```
int insertarSimbolo(char * nombre) {
    int resultado = buscarSimbolo(nombre);
    if(resultado == -1){strcpy(tablaSimbolos[cardinalTabla].id,nombre);
    cardinalTabla++;
    return 0;}
    return resultado;

}
```

Esta función sirve para insertar un nuevo identificador en la tabla de símbolos. Si el símbolo ya existe, retorna el índice donde se encontró. Si no existe, lo inserta en la tabla y aumenta el contador cardinalTabla

```
int insertarSimboloVector(char * nombre) {
    int resultado = buscarSimboloVector(nombre);
    if(resultado == -1){strcpy(vectorDeVariables[cardinalVector].id,nombre);
    cardinalVector++;
    return 0;}
    yyerror("no se puede declarar mas de una vez en un mismo argumento");
    return resultado;

}
```

Esta función sirve para insertar un identificador al vector de símbolos

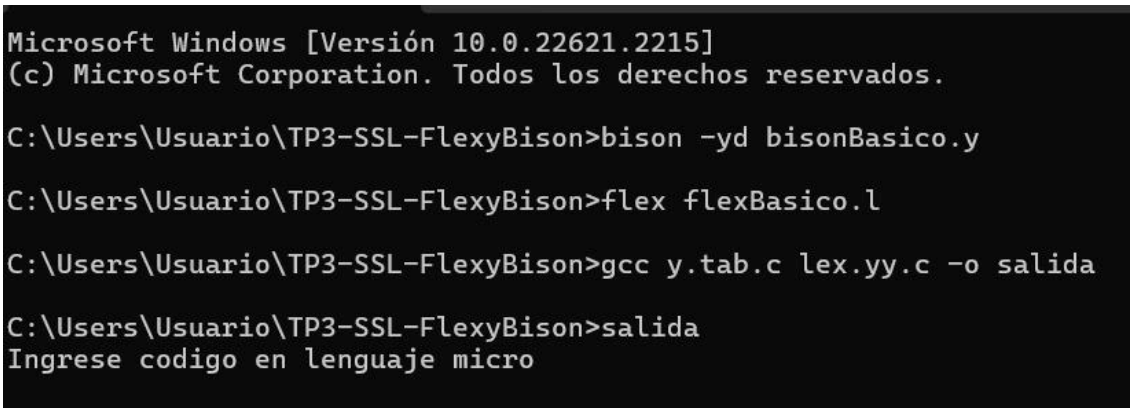
```
int buscarSimboloVector(char* nombre) {
    for (int i = 0; i <cardinalVector; i++) {
        if (strcmp(vectorDeVariables[i].id, nombre) == 0) {
            return i;  // Retorna el índice del símbolo encontrado
        }
    }
    return -1;  // No encontrado
}
```

Esta función busca un identificador en el vector de variables. De encontrar el símbolo retorna su índice, de lo contrario retorna -1.

```
int modificarSimbolo(char* nombre, int nuevoValor) {
    int idx = buscarSimbolo(nombre);
    if (idx != -1) {
        tablaSimbolos[idx].valor = nuevoValor;
        return 0;  // Modificación exitosa
    }
    return -1;  // No encontrado
}
```

Esta función sirve para modificar el valor de un identificador

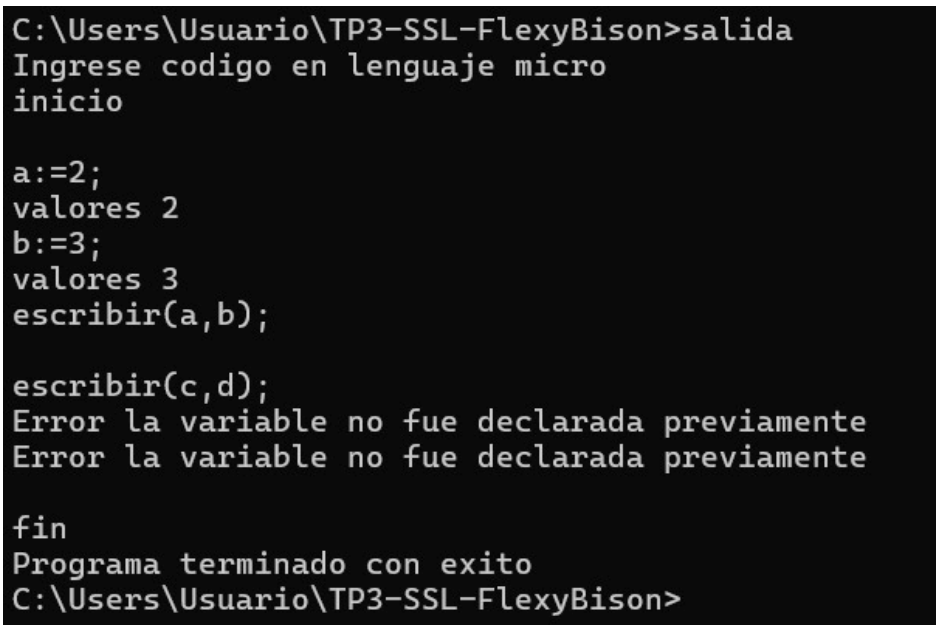
Pantallas demostrando que el programa funciona



Rutinas semánticas implementadas:

La primera rutina semántica que implementamos es la verificación de inicialización de variables. En el siguiente ejemplo demostrativo se muestra como iniciamos las variables “a” y “b” con algunos valores y las utilizamos en “escribir”. El analizador no arroja ningún error ya que las variables contienen valores asignados previamente. Del mismo modo, utilizamos “escribir” con variables que no inicializamos: “c” y “d”, el analizador nos muestra un error personalizado detallando que las variables no fueron inicializadas y por lo tanto no se sabe el valor.

Prueba de que la primera rutina semántica funciona:



Utilizando la misma rutina, nos arroja error si dentro de una expresión utilizamos variables que no inicializamos para asignarla a otra variable. En este caso, “b” no puede ser inicializada ya que no se conoce el valor de “c”.

```
C:\Users\Usuario\TP3-SSL-FlexyBison>salida
Ingrese codigo en lenguaje micro
inicio

a:=1;
valores 1
b:=a+c;
Error la variable no fue declarada previamente

fin
Programa terminado con exito
C:\Users\Usuario\TP3-SSL-FlexyBison>
```

La segunda rutina semántica que implementamos es la verificación de nombres de variables dentro de los argumentos de **leer**, para evitar que se sobreescriba en un mismo id al invocar dicha función.

```
C:\Users\Usuario\TP3-SSL-FlexyBison>salida
Ingrese codigo en lenguaje micro
inicio

leer(a,b);

leer(c,c);
Error no se puede declarar mas de una vez en un mismo argumento

fin
Programa terminado con exito
C:\Users\Usuario\TP3-SSL-FlexyBison>
```

Comandos para iniciar programa:

```
Microsoft Windows [Versión 10.0.22621.2215]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Usuario\TP3-SSL-FlexyBison>bison -yd bisonBasico.y

C:\Users\Usuario\TP3-SSL-FlexyBison>flex flexBasico.l

C:\Users\Usuario\TP3-SSL-FlexyBison>gcc y.tab.c lex.yy.c -o salida

C:\Users\Usuario\TP3-SSL-FlexyBison>salida
Ingrese codigo en lenguaje micro
```