UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Ingeniería en sistemas de Información
Sintaxis y semántica de los lenguajes – 2023

TURNO: Noche CURSO: K2055

DOCENTE A CARGO: Ing. Roxana Leituz
AYUDANTE:

TRABAJO PRÁCTICO TEÓRICO N°3
"Compilador Micro ASA Bison-Flex"

ÁREA TEMÁTICA: Proceso de compilación

GRUPO N° 30
Rodríguez Lucas Ariel
Golato Barcia Ivan Nahuel
Sayago Pablo
Rabahia Maron Leonel
Schinca Mauro

FECHA DE VENCIMIENTO: 17/11/2023
FECHA DE PRESENTACIÓN: 17/11/2023
FECHA DE DEVOLUCIÓN: __/__/__

CALIFICACIÓN: _____ FIRMA PROFESOR: _____

Compilación del programa Compilador Micro:

Para compilar el programa vamos a hacerlo por medio de un archivo MakeFile que contiene el siguiente código:

```
$(BIN): $(OBJ)
    $(CC) $(OBJ) -o $(BIN) $(CFLAGS) $(LFLAGS)

test: $(BIN)
    $(ECHO) Programa micro valido:
    $(BIN) < ./testcases/programaOK.txt
    $(ECHO) ------------------------------------------------------------------------------
    $(ECHO) Programa micro con identificador sin declarar:
    $(BIN) < ./testcases/programaIdentificadorNoDeclarado.txt
    $(ECHO) ------------------------------------------------------------------------------
    $(ECHO) Programa micro leer identificador que es palabra reservada(Error bison):
    $(BIN) < ./testcases/programaIdentificadorEsPalabraReservada.txt
    $(ECHO) ------------------------------------------------------------------------------
    $(ECHO) Programa micro con simbolos no reconocidos:
    $(BIN) < ./testcases/programaSimboloNoReconocido.txt
    $(ECHO) ------------------------------------------------------------------------------
    $(ECHO) Programa micro con asignacion de una variable ya asignada:
    $(BIN) < ./testcases/programaIdentificadorYaDeclarado.txt
    $(ECHO) ------------------------------------------------------------------------------
    $(ECHO) Programa micro con identificador con longitud mayor a 32:
    $(BIN) < ./testcases/programaIdentificadorMayorA32.txt
    $(ECHO) ------------------------------------------------------------------------------
    $(ECHO) Programa micro sentencia sin punto y coma(Error bison):
    $(BIN) < ./testcases/programaSentenciaSinPuntoYComa.txt
    $(ECHO) ------------------------------------------------------------------------------
    $(ECHO) Programa micro leer una contstante(Error bison):
    $(BIN) < ./testcases/programaLeerConstante.txt

all: $(BIN)
```

```
y.tab.c:
    bison -dy parser.y

lex.yy.c:
    flex Scanner.l

y.tab.o:lex.yy.c y.tab.c estructura.h
    $(CC) -c y.tab.c lex.yy.c $(CFLAGS)

clean:
    $(RM) *.o  *.exe lex.yy.c y.tab.c y.tab.h
```

Para realizar la compilación solo basta con abrir una consola en la ubicación de nuestros archivos y ejecutar el comando **make** seguido de la tecla ENTER, como se ve en la siguiente imagen:

```
PS C:\Users\PC1\Documents\Facultad\SSL\SSl-trabajos\CompiladorMicro> make
flex Scanner.l
bison -dy parser.y
gcc -c y.tab.c lex.yy.c -std=c18
```

Esto es equivalente a los comandos flex y bison que aparecen debajo de la ejecución del **make,** así este comando se encarga tanto de compilar los archivos .l e .y en archivos .c y .h como de compilar el ejecutable.
Así se crea “compilador.exe” que recibe su entrada de un .txt con el programa micro desde “stdin”.

Pero el MakeFile no solo nos permite realizar la compilación del programa por medio del comando **make**, sino que también podemos realizar una limpieza de los archivos .o, .exe y los archivos .c y .h de flex y bison por medio del comando **make clean**, esto en el caso que sea necesario realizar algún cambio y volverlo a compilar de una manera rápida (adaptado para funcionar tanto en Windows como Linux).

```
PS C:\Users\PC1\Documents\Facultad\SSL\SSl-trabajos\CompiladorMicroDescendente> make clean
rm -rf *.o compiladorMicro.exe
PS C:\Users\PC1\Documents\Facultad\SSL\SSl-trabajos\CompiladorMicroDescendente>
```

Testeo:

Para realizar el testeo de un posible programa en lenguaje micro hemos agregado el comando **make test** que nos permite verificar el funcionamiento del código en distintos casos de prueba del lenguaje micro (tanto felices como alternativos), como vemos a continuación:

```
Programa micro valido:
compilador.exe < ./testcases/programaOK.txt
El resultado de los argumentos en escribir es 1
Se guardo con exito 1 en el ID variable1
Se guardo con exito 2 en el ID variable2
El resultado de los argumentos en escribir es 3,2
Leo los identificadores ejemplo1,ejemplo2,ejemplo3
Fin del programa
--------------------------------------------------------------------------------
Programa micro con identificador sin declarar:
compilador.exe < ./testcases/programaIdentificadorNoDeclarado.txt
Se guardo con exito 2 en el ID variable2
Leo los identificadores ejemploDeArgumento
Error el identificador var no esta declarado
--------------------------------------------------------------------------------
Programa micro leer identificador que es palabra reservada(Error bison):
compilador.exe < ./testcases/programaIdentificadorEsPalabraReservada.txt
Se guardo con exito 2 en el ID a
Error en parser: syntax error
--------------------------------------------------------------------------------
Programa micro con simbolos no reconocidos:
compilador.exe < ./testcases/programaSimboloNoReconocido.txt
Se guardo con exito 2 en el ID a
Error el simbolo " no pertenece al lenguaje
--------------------------------------------------------------------------------
Programa micro con asignacion de una variable ya asignada:
compilador.exe < ./testcases/programaIdentificadorYaDeclarado.txt
Se guardo con exito 2 en el ID a
Error el identificador a ya esta en uso o es una palabra resevada
--------------------------------------------------------------------------------
Programa micro con identificador con longitud mayor a 32:
compilador.exe < ./testcases/programaIdentificadorMayorA32.txt
Error el ID qwertyuiopasdfghjklzxcvbnmqwertyu es demasiado largo
--------------------------------------------------------------------------------
Programa micro sentencia sin punto y coma(Error bison):
compilador.exe < ./testcases/programaSentenciaSinPuntoYComa.txt
Error en parser: syntax error
--------------------------------------------------------------------------------
Programa micro leer una contstante(Error bison):
compilador.exe < ./testcases/programaLeerConstante.txt
Se guardo con exito 1 en el ID variable1
Error en parser: syntax error
```

A lo largo del programa se va imprimiendo por consola el resultado del análisis de cada línea del programa micro.

Flex:

En el archivo scanner.l podemos encontrar todo el código relacionado al scanner de nuestro compilador

```
%option noyywrap
%{
#include "./y.tab.h"
#include <string.h>
void verificarLongitudId(int);///tipos
%}
CONSTANTE       [0-9]+
IDENTIFICADOR   [A-Za-z][A-Za-z0-9]*

%%
"inicio" {return INICIO;}
"fin" {return FIN;}
"leer" {return LEER;}
"escribir" {return ESCRIBIR;}
"(" {return PARENTESISIZQ;}
")" {return PARENTESISDER;}
";" {return PUNTOYCOMA;}
"," {return COMA;}
":=" {return ASIGNACION;}
"+" {return SUMA;}
"-" {return RESTA;}
"FDT" {return FDT;}
{CONSTANTE} {yylval.num=atoi(yytext);return CONSTANTE;}
{IDENTIFICADOR} {verificarLongitudId(yyleng);yylval.palabra=strdup(yytext);return IDENTIFICADOR;}
\s|\n|\t|" " {}
. {printf("Error el simbolo %s no pertenece al lenguaje\n",yytext);exit(0);}
%%

void verificarLongitudId(int longitudID)
{
    if(longitudID>32)
    {
        printf("Error el ID %s es demasiado largo\n",yytext);
        exit(0);
    }
}
```

Como sabemos los archivos flex se dividen principalmente en tres partes definiciones, reglas y código c.

En las definiciones utilizamos las regex para definir los conjuntos de números en el caso de las constantes y números con letras en el caso de los identificadores, esto es para simplificar su uso a lo largo del programa.

En las reglas definimos los lexemas que deben ser reconocido con su categoría léxica que es devuelta al parser, estas categorías estaban previamente definidas en la gramática léxica.

En ciertos casos hicimos algo más que devolver solo el token, por ejemplo en el caso de las CONSTANTES no solo hay que devolver el token al parser sino que se debe convertir de texto, que está en la variable yytext, a entero por medio de la función atoi.

Por otro lado, en los identificadores se utilizó la función strdup, ya que nos permite crear una copia del puntero al string en un nuevo espacio de memoria. Además, de asociar el valor del ID a un token se realiza una verificación de longitud del nombre del ID, ya que micro no permite tener más de 32 caracteres, esto lo verifica en la función verificarLongitudId en la parte de código c del archivo.

Bison:

En el archivo parser.y podemos encontrar todo el código relacionado al parser AAS de nuestro compilador .Principalmente es una transcripción de la gramática sintáctica de micro con leves cambios y agregando las rutinas semánticas correspondientes.

```
%{
#include "estructura.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
extern int yylex(void);
extern void yyerror(char*);

%}

%token INICIO FIN LEER ESCRIBIR IDENTIFICADOR CONSTANTE PARENTESISIZQ PARENTESISDER PUNTOYCOMA COMA ASIGNACION SUMA RESTA FDT

%union{
    char *palabra;
    int num;
}

%type <num> CONSTANTE primaria expresion
%type <palabra> listaID  IDENTIFICADOR listaExpresiones
%%

objetivo: program FDT {printf("Fin del programa\n");return 0;}
;

program:INICIO  listaSentencias FIN
;

listaSentencias: sentencia listaSentencias
| sentencia
;
```

```
sentencia:IDENTIFICADOR ASIGNACION expresion PUNTOYCOMA {if(!existeID($1,cont))
                                                        {
                                                        guardar($1,$3,++cont);
                                                        printf("Se guardo con exito %d en el ID %s\n",$3,$1);
                                                        }
                                                        else{
                                                        printf("Error el identificador %s ya esta en uso o es una palabra resevada\n",$1);
                                                        exit(0);
                                                        };
                                                        }
| LEER PARENTESISIZQ listaID PARENTESISDER PUNTOYCOMA {printf("Leo los identificadores %s\n",$3);
                                                      }
| ESCRIBIR PARENTESISIZQ listaExpresiones PARENTESISDER PUNTOYCOMA{printf("El resultado de los argumentos en escribir es %s\n",$3);
                                                                  }
;

listaID:IDENTIFICADOR {$$=$1;}
| IDENTIFICADOR COMA listaID{$$=strcat(strcat($1,","),$3);}
;

listaExpresiones:expresion {char aux[10];
                  sprintf(aux, "%d", $1);
                  $$=aux;}
| expresion COMA listaExpresiones{char aux[10];
                       sprintf(aux, "%d", $1);
                    char *aux2 = aux;
                    $$=strcat(strcat(aux2,","),$3);}
;
```

```
expresion:primaria {$$=$1;}
| primaria RESTA primaria{$$=$1-$3;}
| primaria SUMA primaria{$$=$1+$3;}
;

primaria:CONSTANTE {$$=$1;}
| IDENTIFICADOR {int aux = existeID($1,cont);
        if(aux)
                {
                $$=valorID(aux);
                }
                else
                {
                printf("Error el identificador %s no esta declarado\n",$1);
                exit(0);
                };
                }
| PARENTESISIZQ expresion PARENTESISDER {$$=$2;}
;

%%

int main()
{
    cont = cargarPalabrasReservadas();
    yyparse();
    return 0;
}

void yyerror(char* p)
{
    fprintf(stderr, "Error en parser: %s\n",p);
}
```

En este archivo se declaran las gramáticas sintácticas en las cuales se puede agregar rutinas semánticas por medio de código c entre { }.

En la parte superior de la primera imagen se ven los tokens que vamos a reconocer, los cuales se declaran por medio de la directiva %token. Por otro lado, le adjuntamos tipos a nuestros tokens como a nuestros no terminales por medio de la declaración de %union para definir los tipos y %type<tipo> para adjuntarlos a nuestros tokens.

Como podemos ver en la primera imagen se utiliza el header estructuras.h que contiene el siguiente código:

```c
#ifndef ESTRUCTURA_H
#define ESTRUCTURA_H
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
...
struct id{
    int valor;
    char *nombre;
}tabla[20];
int cont=0;
int existeID(char *,int);
int cargarPalabrasReservadas();
void guardar(char * ,int  , int );
int valorID(int posicion);

int existeID(char *nombre,int posiciones)...

int cargarPalabrasReservadas()...

void guardar(char * nombre,int valor , int posicion)...

int valorID(int posicion)...
#endif
```

Principalmente el header contiene la tabla de símbolos para nuestras palabras reservadas e ID, además de todas las funciones para manipular la misma.

Rutinas semánticas:

En el siguiente apartado mostramos las rutinas semánticas que implementamos en nuestro compilador micro con lenguaje c.

Control de nuestros identificadores:

Para saber si un identificador de un posible programa micro está declarado y que valor contiene, decidimos utilizar un array de struct llamado “tabla” que puede almacenar el valor y nombre de un ID , como así también puede guardar las palabras reservadas por medio de un procedimiento.

```
sentencia:IDENTIFICADOR ASIGNACION expresion PUNTOYCOMA {if(!existeID($1,cont))
                                                        {
                                                        guardar($1,$3,++cont);
                                                        printf("Se guardo con exito %d en el ID %s\n",$3,$1);
                                                        }
                                                        else{
                                                        printf("Error el identificador %s ya esta en uso o es una palabra resevada\n",$1);
                                                        exit(0);
                                                        };
                                                        }
```

En este caso estamos en una declaración de un ID, así que debemos verificar si ya fue declarado previamente, por medio de la función existeID podemos estar seguro de ello. Sí el ID no está en la tabla se procede a guardarlo por la función guardar, en caso contrario se producirá un error semántico y concluirá el programa.

```
primaria:CONSTANTE {$$=$1;}
| IDENTIFICADOR {int aux = existeID($1,cont);
        if(aux)
                {
                $$=valorID(aux);
                }
                else
                {
                printf("Error el identificador %s no esta declarado\n",$1);
                exit(0);
                };
                }
| PARENTESISIZQ expresion PARENTESISDER {$$=$2;}
;
```

En este caso verificamos que exista el ID para su uso en una expresión, que puede ser una operación. Sí existe buscamos su valor por medio de valorID y se lo asignamos a primarios por medio de $$=valorID() .En caso contrario se producirá un error semántico y concluirá el programa .

Hacer cuentas en escribir:

```
| ESCRIBIR PARENTESISIZQ listaExpresiones PARENTESISDER PUNTOYCOMA{printf("El resultado de los argumentos en escribir es %s\n",$3);
                                                                 }
;

listaID:IDENTIFICADOR {$$=$1;}
| IDENTIFICADOR COMA listaID{$$=strcat(strcat($1,","),$3);}
;

listaExpresiones:expresion {char aux[10];
                  sprintf(aux, "%d", $1);
                  $$=aux;}
| expresion COMA listaExpresiones{char aux[10];
                       sprintf(aux, "%d", $1);
                    char *aux2 = aux;
                    $$=strcat(strcat(aux2,","),$3);}
;

expresion:primaria {$$=$1;}
| primaria RESTA primaria{$$=$1-$3;}
| primaria SUMA primaria{$$=$1+$3;}
;

primaria:CONSTANTE {$$=$1;}
| IDENTIFICADOR {int aux = existeID($1,cont);
        if(aux)
                {
                $$=valorID(aux);
                }
                else
                {
                printf("Error el identificador %s no esta declarado\n",$1);
                exit(0);
                };
                }
| PARENTESISIZQ expresion PARENTESISDER {$$=$2;}
;
```

El código previo muestra el camino desde los no terminales primaria,que definimos como int previamente, hasta la función escribir donde se lee el valor final de la operación y se muestra.

Para realizar esto le dimos valores int a expresión, primaria (por medio de la sentencia $$=un número o una operación) y char* a listaExpresiones (ya que se trata de una lista que puede tener varios valores numéricos) esto con el fin de transportar el valor hacia la función escribir y por último leerlo con la variable $ que bison le asigna al token.

Cantidad de caracteres ID:

```
{CONSTANTE} {yylval.num=atoi(yytext);return CONSTANTE;}
{IDENTIFICADOR} {verificarLongitudId(yyleng);yylval.palabra=strdup(yytext);return IDENTIFICADOR;}
\s|\n|\t|" " {}
. {printf("Error el simbolo %s no pertenece al lenguaje\n",yytext);exit(0);}
%%

void verificarLongitudId(int longitudID)
{
    if(longitudID>32)
    {
        printf("Error el ID %s es demasiado largo\n",yytext);
        exit(0);
    }
}
```

Como mencionamos previamente en el scanner hacemos una comprobación de la longitud de nuestros ID por medio de yyleng ,que nos da la longitud de yytext, y de la función producimos el error si es necesario.