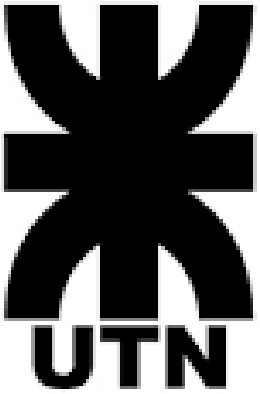
INFORME TP 3 FLEX Y BISON

Materia: SSL

Explicación del pensamiento detrás del trabajo



Fecha de entrega: 17/11/2023

Integrantes Grupo 25:

-CACACE, Guillermo Federico

-CALÓ, Ignacio

-GOMEZ PEREYRA, Manuel Francisco

- MAJER, Cecilia Alejandra

-TROSSERO, Agustín Francisco

Desarollo y manual de usuario

En un inicio, procedimos a escribir el archivo flex para el armado del scanner. En un primer lugar realizamos las declaraciones de los token asignándoles su RegEx correspondiente:

```
%}
DIGITO [0-9]
LETRA [a-zA-Z]
IDENTIFICADOR {LETRA}({LETRA}|{DIGITO})*
constEntera {DIGITO}({DIGITO})*
%%
```

Luego procedimos a definir las reglas léxicas, que constan de definir los distintos operadores, centinelas, palabras reservadas, constantes, identificadores. Nótese que lo encerrado entre llaves es código en C que reaccionará cuando se den los casos del lado izquierdo

```
":=" { return ASIGNACION; }
";" { return PYCOMA; }
"(" { return PARENIZQUIERDO; }
"+" { return SUMA; }
")" { return PARENDERECHO; }
"-" { return RESTA; }
"inicio"          {yylval.reservada = yytext; return (INICIO);}
"fin"             {yylval.reservada = yytext; return (FIN);}
"leer"            {yylval.reservada = yytext; return (LEER);}
"escribir"        {yylval.reservada = yytext; return (ESCRIBIR);}

{constEntera} { yylval.num = atoi(yytext); return CONSTANTE; }
{IDENTIFICADOR} { yylval.cadena = strdup(yytext); return ID; }
[ \t\n]       { /* Ignorar espacios en blanco y saltos de línea */ }
.             { fprintf(stderr, "Caracter inesperado: %s\n", yytext);yyerror("Error lexico"); }
%%
```

yylval.<type> y *yyerror();* Son funciones que tendrán efecto al combinarlos con el bison.

En el caso del Bison:

declaramos los tipos asociados a los token viables en la estructura sintáctica

```
%union{
char* cadena;
int num;
char* reservada;
}
%token ASIGNACION PYCOMA SUMA RESTA PARENIZQUIERDO PARENDERECHO
%token <cadena> ID
%token <num> CONSTANTE
%type <num> expresion
%type <num> primaria
%token <reservada> INICIO FIN LEER ESCRIBIR
```

Luego definimos la GIC para estructurar la sintaxis del programa ayudándonos de los noTerminales que al derivarse irán resolviendo las reglas sintácticas. Luego, las partes encerradas entre llaves se declaran la reacción tal cual como en el flex ante ciertas condiciones en código C, que correrán errores, rutinas semánticas e impresiones de pantalla.

```
programa: INICIO sentencias FIN
;
sentencias: sentencias sentencia
|sentencia
;
```

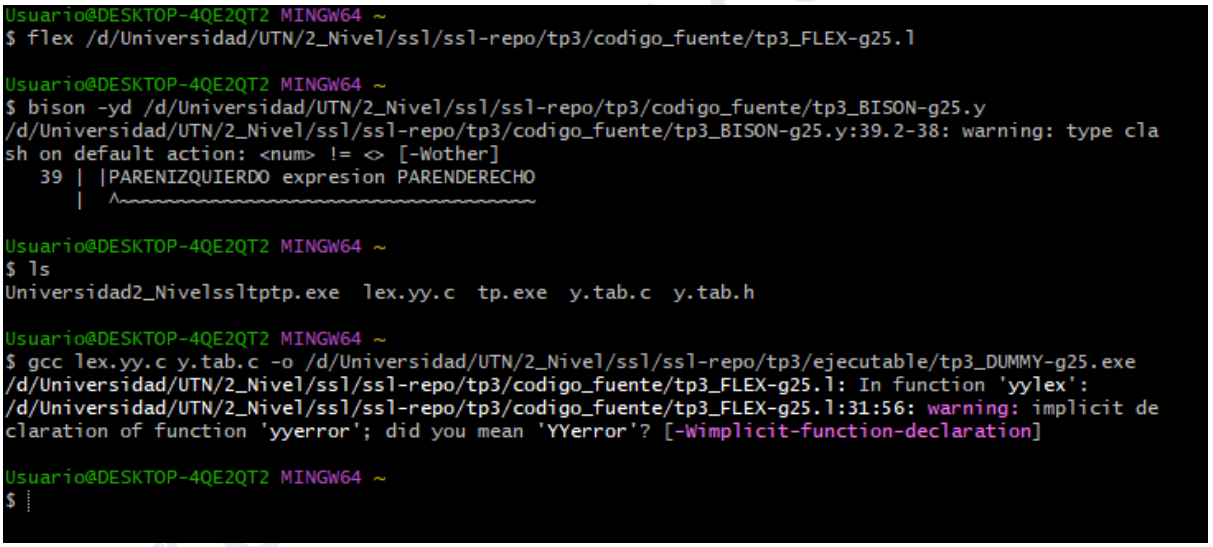
```
sentencia: ID {printf("\nRUTINA SEMANTICA:\tLa longitud del ID es: %d",yyleng);if(yyleng>33)yyerror("La longitud del ID es mayor a la permitida");} ASIGNACION expresion PYCOMA
|LEER expresion PYCOMA
|ESCRIBIR expresion PYCOMA
;
expresion: primaria
|expresion operadorAditivo primaria {$$ = $1 + $3;printf("\nRUTINA SEMANTICA:\tEl resultado es %d \n",$$);}
|expresion operadorResta primaria {$$ = $1 - $3;printf("\nRUTINA SEMANTICA:\tEl resultado es %d \n",$$);}
|error { yyerror("Error en la expresión"); }
;
primaria:ID {printf("\nRUTINA SEMANTICA:\tLa longutid del ID es: %d",yyleng);if(yyleng>33)yyerror("La longitud del ID es mayor a la permitida");}
|CONSTANTE {printf("\nRUTINA SEMANTICA:\tValor de la constante: %d\n",$1);}
|PARENIZQUIERDO expresion PARENDERECHO
;
operadorAditivo: SUMA
;
operadorResta:RESTA
;
```

En la última parte declaramos las funciones que harán efectiva la parte de C del programa.

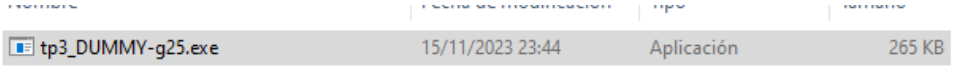
```
%%
int main(){
        yyparse();
```

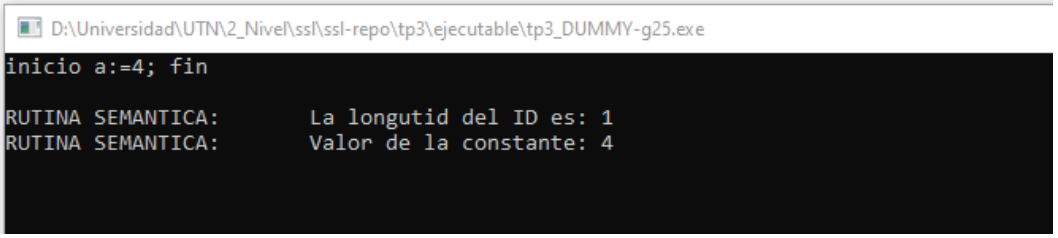
```
}
void yyerror(char *s){
	fprintf(stderr,"\nMotivo del error: %s\n",s);
	exit(1);
}
int yywrap(){
	return 1;
}
```

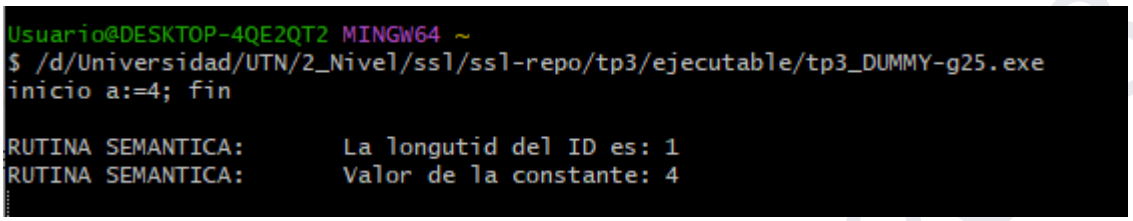
Una vez escritos los programas se proceden a compilarlos en flex y en bison, que nos producirán los archivos necesarios para luego poder compilar el programa total en C. Utilizamos ***MINGW64*** para realizar las líneas de comandos:



Una vez obtenido el ejecutable, lo abrimos ya sea por click directo o por la misma terminal en ***MINGW64*** y podemos escribir código el cuál analizará:



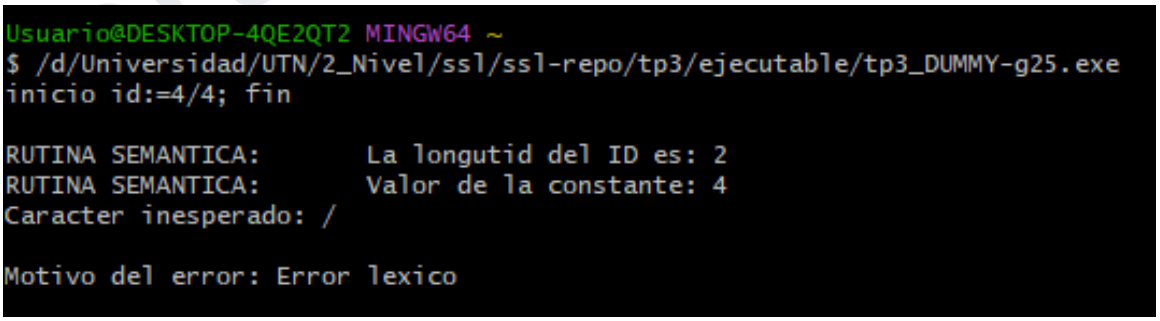




Pantallas de Funcionamiento

Primeramente, armamos la estructura del lenguaje MICRO Fisher, reconociendo y denotando todos sus tokens en Flex/Lex. Separamos en DÍGITO, LETRA, IDENTIFICADOR y constEntera, implementandolos como REGEX; además de luego introducir los tokens necesarios de puntuación y operadores, junto al ‘inicio’ y ‘fin’ del programa. Aquí agregamos el mensaje de “Error léxico”, en el caso de que se introduzca al flujo un lexema no perteneciente al lenguaje.

Caso de prueba: Una división (el operador ‘/’ no pertenece al lenguaje)



Posteriormente, en Bison definimos las estructuras válidas para un ‘programa’, el cual está compuesto por INICIO, sentencias y FIN. Describimos cuales son todas las sentencias válidas, y su estructura debida.

Personalizamos los errores sintácticos, como por ejemplo, un ID de más de 32 caracteres:

```
Usuario@DESKTOP-4QE2QT2 MINGW64 ~
$ /d/Universidad/UTN/2_Nivel/ssl/ssl-repo/tp3/ejecutable/tp3_DUMMY-g25.exe
inicio aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa:=4+5+2; fin

RUTINA SEMANTICA:       La longutid del ID es: 53
Motivo del error: La longitud del ID es mayor a la permitida
```

Además, si la ESTRUCTURA de las declaraciones es errónea, muestra por pantalla un error sintáctico. En este ejemplo, decidimos no terminar la sentencia con un ‘;’ para que sea errónea:

```
Usuario@DESKTOP-4QE2QT2 MINGW64 ~
$ /d/Universidad/UTN/2_Nivel/ssl/ssl-repo/tp3/ejecutable/tp3_DUMMY-g25.exe
inicio id:=4 fin

RUTINA SEMANTICA:       La longutid del ID es: 2
RUTINA SEMANTICA:       Valor de la constante: 4

Motivo del error: syntax error
```

Luego elegimos como rutinas semánticas, la devolución de longitud y valor de los identificadores introducidos en el programa, y además, el resultado de las operaciones (expresiones) que se ingresan. A continuación ejemplos de diferentes casos:

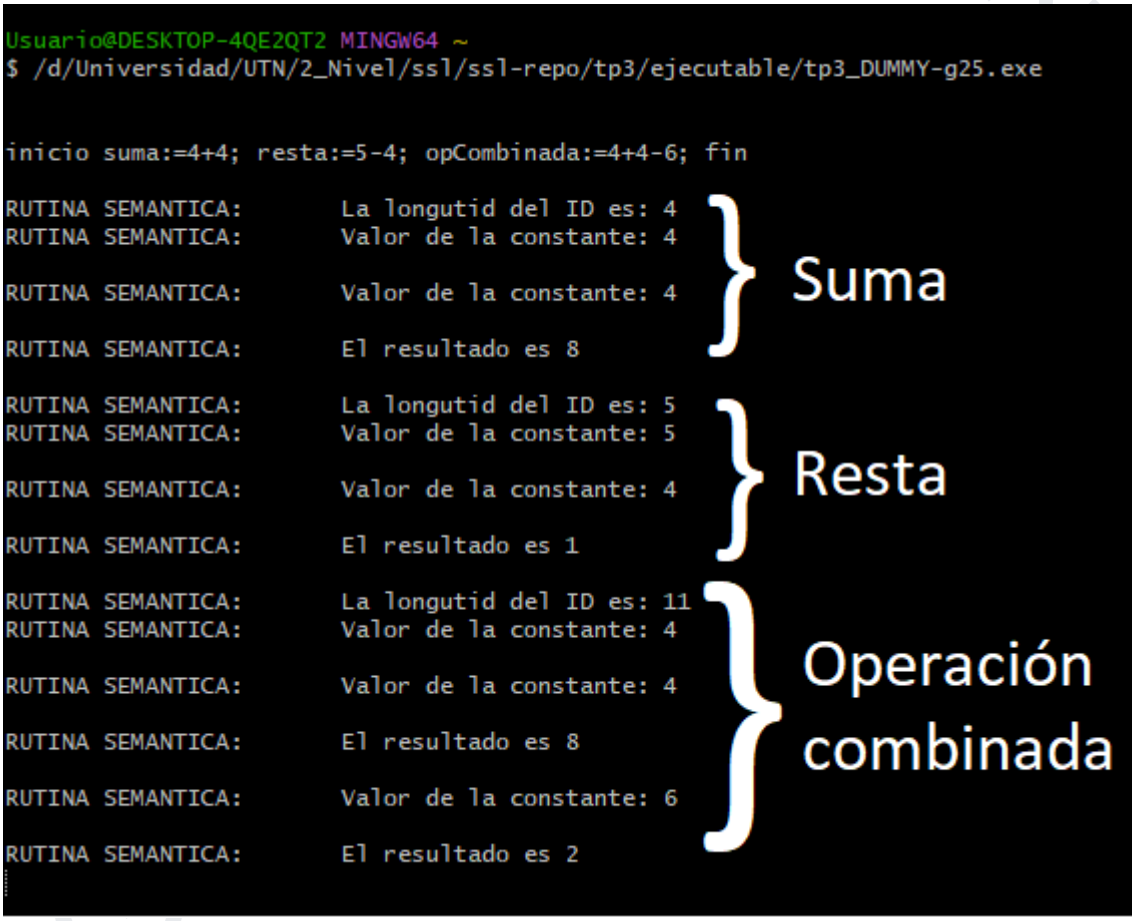
CASO 1

Ingresamos dos ID, uno llamado ‘id’, de longitud 2 y valor 444, y otro llamado ‘a’, de longitud 1 y valor 8.

```
Usuario@DESKTOP-4QE2QT2 MINGW64 ~
$ /d/Universidad/UTN/2_Nivel/ssl/ssl-repo/tp3/ejecutable/tp3_DUMMY-g25.exe



inicio id:=444; a:=8; fin

RUTINA SEMANTICA:       La longutid del ID es: 2
RUTINA SEMANTICA:       Valor de la constante: 444

RUTINA SEMANTICA:       La longutid del ID es: 1
RUTINA SEMANTICA:       Valor de la constante: 8
```

CASO 2

Ingresamos tres operaciones diferentes, una suma, una resta y una combinación de ambas. Las rutinas semánticas nos devuelven la longitud de los ID a los que están asignadas esas expresiones, los valores de cada número detectado en la operación, y el resultado de la misma. En el caso de una combinada, también nos va arrojando la resolución parcialmente de izquierda a derecha.



CASO 3

No implementamos las rutinas semánticas de verificar la declaración previa , ya que no tenemos una pila donde almacenar estos datos temporalmente, pero sí es válido desde un punto tanto léxico como sintáctico.

```
Usuario@DESKTOP-4QE2QT2 MINGW64 ~
$ /d/Universidad/UTN/2_Nivel/ssl/ssl-repo/tp3/ejecutable/tp3_DUMMY-g25.exe


inicio leer(4); escribir(4+5); fin

RUTINA SEMANTICA:       Valor de la constante: 4

RUTINA SEMANTICA:       Valor de la constante: 4

RUTINA SEMANTICA:       Valor de la constante: 5

RUTINA SEMANTICA:       El resultado es 9
```