MEMORIA PROYECTO MINIC

COMPILADORES

Alumnos: Rafael Guillén García

Grupo: 2.4

Convocatoria: Junio de 2025

ÍNDICE

1. MANUAL DE USUARIO

Para la correcta ejecución del programa el usuario debe seguir los siguientes pasos:

1. Primero, debe descargar la carpeta comprimida “Proyecto miniC.zip”, y descomprimirla dentro de su equipo.

2. Asegúrese de que, dentro de la carpeta, se encuentra un ejecutable llamado “miniC”. Si no lo encuentra, abra una terminal dentro de la carpeta, y ejecute el comando “make”, el cual generará dicho ejecutable a partir de los ficheros de código fuente.

3. Para utilizar la aplicación, debe disponer de, al menos, un programa escrito en lenguaje miniC (un archivo con la extensión “.mc”) en su equipo. Puede usar los que hay en la carpeta “Archivos prueba”.

4. Para compilar un programa miniC usando la aplicación, basta con abrir una terminal dentro de la carpeta que ha descomprimido antes, y ejecutar el comando “./miniC [nombre_del_programa].mc > [nombre_de_la_salida].s”.

5. Ejecute el archivo “.s” generado como salida en cualquier intérprete de código ensamblador (por ejemplo, MARS o Spim). Si utiliza Spim, debe ejecutar el comando “./spim –file [nombre_del_programa].s”.

2. FUNCIONES PRINCIPALES

En este apartado, se mencionan las distintas funciones que hemos utilizado para la implementación de nuestro compilador, así como un resumen de su funcionalidad.

`yyparse()`: Método en el método principal del programa. Se utiliza para obtener los tokens de la entrada, y devuelve un entero (0 o 1) indicando si el analizador léxico ha finalizado su ejecución exitosamente, o si ha habido algún error.

`yyerror()`: Método que imprime un mensaje de error en caso de error sintáctico, indicando el token que causó el error y la línea de la entrada en la que se encuentra.

`creaLS()`: Método que crea una lista de símbolos nueva.

`liberaLS(Lista lista)`: Método que libera la lista de símbolos pasada como parámetro.

`insertaLS(Lista lista, PosicionLista p, Simbolo s)`: Método que crea un nuevo nodo con el símbolo "s", e inserta el nodo en la posición "p" de la lista de símbolos "lista".

`suprimeLS(Lista lista, PosicionLista p)`: Método que elimina el símbolo de la posición "p" de la lista de símbolos "lista".

`recuperaLS(Lista lista, PosicionLista p)`: Método que devuelve el símbolo que se encuentra en la posición "p" de la lista de símbolos "lista".

`buscaLS(Lista lista, char *nombre)`: Método que busca en la lista de símbolos "lista" un símbolo cuyo nombre coincida con "nombre".

`asignaLS(Lista lista, PosicionLista p, Simbolo s)`: Método que busca el nodo que se encuentra en la posición "p" de la lista de símbolos "lista", y cambia su símbolo por el símbolo "s".

`longitudLS(Lista lista)`: Método que devuelve la longitud de la lista de símbolos "lista".

`inicioLS(Lista lista)`: Método que devuelve el nodo de la cabecera de la lista de símbolos "lista".

finalLS(Lista lista): Método que devuelve el último nodo de la lista de símbolos “lista”.

siguienteLS(Lista lista, PosicionLista p): Método que devuelve el nodo siguiente del que se encuentra en la posición “p” de la lista de símbolos “lista”.

perteneceTablaS(Lista l, char * nombre): Método que devuelve un entero indicando si el símbolo de nombre “nombre” pertenece a la tabla de símbolos.

insertaTablaIdentificador(Lista l, char *nombre, Tipo tipo): Método que inserta un símbolo con el nombre y el tipo pasados como parámetro en la tabla de símbolos.

insertaTablaString(Lista l, char *nombre, Tipo tipo, int valor): Método que inserta un símbolo con el nombre, tipo y valor pasados como parámetro en la tabla de símbolos.

esConstante(Lista l, char *nombre): Método que verifica si el símbolo de la lista de símbolos “l” cuyo nombre es “nombre” es de tipo constante.

imprimirTablaLS(Lista l): Método que imprime una serie de cadenas correspondientes al segmento de datos de un programa en ensamblador de MIPS utilizando el contenido de la lista de símbolos “l”.

creaLC(): método que crea una lista de código nueva.

liberaLC(ListaC codigo): Método que libera la lista de código pasada como parámetro.

insertaLC(ListaC codigo, PosicionListaC p, Operacion o): Método que inserta en la posición “p” de la lista de código “codigo” un nodo con la operación “o”.

recuperaLC(ListaC codigo, PosicionListaC p): Método que devuelve la operación de la posición siguiente a la posición “p” de la lista de código “codigo”.

buscaLC(ListaC codigo, PosicionListaC p, char *clave, Campo campo): Método que devuelve la operación de la posición “p” de la lista de código “codigo” o posterior, y selecciona la información del campo “campo” de dicha operación. Solo devuelve la posición si la información coincide con la cadena “clave”. En caso contrario, busca otra posición.

`asignaLC(ListaC codigo, PosicionListaC p, Operacion o):` Método que busca el nodo de la posición "p" de la lista de código "codigo", y sustituye la operación que contiene por "o".

`longitudLC(ListaC codigo):` Método que devuelve la longitud de la lista de código pasada como parámetro.

`inicioLC(ListaC codigo):` Método que devuelve el nodo de la cabecera de la lista de código "codigo".

`finalLC(ListaC codigo):` Método que devuelve el último nodo de la lista de código "codigo".

`concatenaLC(ListaC codigo1, ListaC codigo2):` Método que concatena las dos listas de código pasadas como parámetros.

`siguienteLC(ListaC codigo, PosicionListaC p):` Método que devuelve el nodo siguiente al de la posición "p" de la lista de código "codigo".

`guardaResLC(ListaC codigo, char *res):` Método que guarda un resultado obtenido en la lista de código "codigo".

`recuperaResLC(ListaC codigo):` Método que devuelve el resultado de la lista de código "codigo".

`imprimirLC(ListaC l):` Método que imprime por pantalla un listado con las operaciones correspondiente al segmento de código de un programa en ensamblador de MIPS utilizando el contenido de la lista de código "codigo".

`iniciaRegistros():` Método que inicializa los registros disponible s para utilizar cvon el valor 0.

`queRegistro():` Método que devuelve el registro de menor número que no esté ocupado.

`liberaRegistro(char *registro):` método que libera el registro pasado como parámetro.

`concatenar(char* cad1, char* cad2):` Método que concatena dos cadenas de caracteres pasadas como parámetro.

`queStr():` Método que devuelve las dos cadenas concatenadas.

`obtenerEtiqueta():` Método que devuelve una etiqueta.

3. ESTRUCTURAS DE DATOS

Para la implementación de nuestro compilador, hemos hecho uso de diversas estructuras de datos definidas con %union y struct:

```
%union{
char *lexema;
ListaC codigo;
}
```

Esta unión significa que los tokens y no terminales pueden devolver o almacenar

- Un char* lexema
- Una lista de código llamada código

```
typedef struct Nodo {
char *nombre;
Tipo tipo;
int valor;
} Simbolo;
```

Este struct representa cada uno de los símbolos que acepta nuestro compilador. Cada uno de esos símbolos estará formado por tres variables: un nombre, un tipo (que puede ser una variable, una constante, o una cadena de caracteres), y un valor, que representa el valor actual del símbolo en un momento determinado.

```
struct PosicionListaRep {
Simbolo dato;
struct PosicionListaRep *sig;
};
```

Este struct representa, como su nombre indica, una posicñon en una lista. En este caso concreto, lo hemos utlizado en listaSimbolos.c, por lo que representa un lugar en dicha lista de símbolos. Está formado por el dato que osupa esa posición, que es de tipo Simbolo, y por un apuntador al siguiente elemento de la lista, que será nulo si el elemento actual es el último de la lista de símbolos.

```
struct ListaRep {
PosicionLista cabecera;
PosicionLista ultimo;
int n;
};
```

Este struct representa, como su nombre también indica, una lista de elementos. En este caso, también se ha usado en listaSimbolos.c, por lo que representa a esa lista de símbolos en su totalidad. Está formado por dos elementos de tipo PosicionLista que representan al primer y al último elemento de la lista, y por una variable de tipo entero que almacena la longitud de la lista.

```
typedef struct {
char * op;
char * res;
char * arg1;
char * arg2;
} Operacion;
```

Este struct representa cada una de las operaciones que vamos a poder realizar en nuestro compilador. Está formado por cuatro cadenas de caracteres: la primera es el identificador de la operación a realizar (“sw”, por ejemplo), y las otras tres representan los operandos con los que trabaja la operación en cuestión, siendo res donde se almacena el resultado de la operación, y arg1 y arg2 los que sirven como datos para realizar una operación, como puede ser una suma con “add”.

```
struct PosicionListaCRep {
Operacion dato;
struct PosicionListaCRep *sig;
};
```

Este struct representa una posición en una lista, pero, en este caso, se refiere a la lista de código, de ahí la “C” en el nombre. Está formado por una variable de tipo Operación que representa un dato, y un puntero que apunta al siguiente elemento de la lista de código.

```
struct ListaCRep {
PosicionListaC cabecera;
PosicionListaC ultimo;
int n;
char *res;
};
```

Este struct representa a una lista de código en su totalidad. Semejante a la lista de símbolos, este tipo también almacena dos variables de tipo PosicionListaC con los elementos primero y último de la lista, y una variable entera con su longitud. Además, almacenamos también una cadena de caracteres que representa al primer argumento de una operación.

4. EJEMPLO DE FUNCIONAMIENTO

Se realiza la siguiente llamada:

./miniC ./ficherosDePrueba/pruebaTotal.mc > salida.S

Prueba.mc contiene:

```
prueba() {
const int a = 0, b = 0;
var int c;
print ("Inicio del programa\n");
c = 5+2-2;
if (a) print ("a","\n");
else if (b) print ("No a y b\n");
else while (c) {
print ("c = ",c,"\n");
c = c-2+1;
}
print ("Final","\n");
}
```

salida.S contiene:

```
##################
# Seccion de datos
.data
_a: .word 0
_b: .word 0
_c: .word 0
$str0: .asciiz "Inicio del programa\n"
$str1: .asciiz "a"
$str2: .asciiz "\n"
$str3: .asciiz "No a y b\n"
$str4: .asciiz "c = "
$str5: .asciiz "\n"
$str6: .asciiz "Final"
$str7: .asciiz "\n"
##################
# Seccion de codigo
.text
.globl main
main:
li $t0, 0
sw $t0, _a
li $t0, 0
sw $t0, _b
li $t0, 0
la $a0, $str0
li $v0, 4
syscall
li $t0, 5
```

```
li $t1, 2
add $t2, $t0, $t1
li $t0, 2
sub $t1, $t2, $t0
sw $t1, _c
lw $t0, _a
beqz $t0, $l5
la $a0, $str1
li $v0, 4
syscall
la $a0, $str2
li $v0, 4
syscall
b $l6
$l5:
lw $t1, _b
beqz $t1, $l3
la $a0, $str3
li $v0, 4
syscall
b $l4
$l3:
$l1:
lw $t2, _c
beqz $t2, $l2
la $a0, $str4
li $v0, 4
syscall
lw $t3, _c
```

```
li $v0, 1
move $a0, $t3
syscall
la $a0, $str5
li $v0, 4
syscall
lw $t3, _c
li $t4, 2
sub $t5, $t3, $t4
li $t3, 1
add $t4, $t5, $t3
sw $t4, _c
b $l1
$l2:
$l4:
$l6:
la $a0, $str6
li $v0, 4
syscall
la $a0, $str7
li $v0, 4
syscall
##################
# FIN
li $v0, 10
syscall
```

5. CONCLUSIÓN

En conclusión, podemos decir que la realización de este proyecto ha conseguido que comprendamos mucho mejor el funcionamiento de los compiladores y los contenidos de la asignatura. Creemos que el tiempo invertido en la implementación de los distintos ficheros ha valido la pena, aunque ha sido bastante difícil el correcto funcionamiento del compilador.

6. CORRECCIONES

1. Solo se imprime la línea del fallo cuando se entra en el modo pánico.

    Lo hemos arreglado añadiendo la opción “lineno” al error del identificador y del entero

2. No se reconocen identificadores que comienzan por "_".

    Hemos añadido a la expresión regular _| :

    *_|{letra}({letra}|{digito}|_)**

3. No se detectan comentarios multilínea sin cerrar.

    Hemos arreglado la expresión regular:

    "/"([^*]|*+[^*/])**+"/"*

4. Da/Muestra código cuando se detectan errores.

    Lo hemos arreglado con externalizando la variable errores del flex sustituyéndola por la variable erroresY en el bison (lo hicimos así en un principio, pero no funcionaba)

5. No se explica por qué se utiliza %expect 1 en la gramática; es necesario ofrecer una explicación detallada y no limitarse a indicar que es para evitar un warning.

    La gramática genera un conflicto S/R con las sentencias if y if-else, Bison por defecto desplaza, asociando el else con el if más cercano, esto es lo que queremos y es por eso que ponemos el %expect 1, para evitar un warning

6. No se imprimen errores semánticos (solo se incrementa el contador). Se deberían de imprimir.

    Tras la revisión del código nos dimos cuenta que mientras realizábamos el proyecto comentamos los prints de los errores semánticos, ademas hemos creado una nueva función para mostrar los errores en pantalla (errorSem())

7. No se liberan listas de código en const_list; cada vez que se concatenan dos listas, hay que liberar la segunda.

    Lo hemos arreglado añadiendo un liberaLC() de la segunda lista después del último concatenaLC() de const_list

8. Se necesitan más casos de prueba que muestren tanto el funcionamiento correcto de la gramática completa como el manejo de todos los errores.

    Hemos añadido otro caso prueba para el manejo de errores más el proporcionado por usted para comprobar la cuarta corrección.