



CURSO 2021-2022

“MINI C”

**Traducción de miniC acódigo
ensamblador de MIPS**

COMPILADORES

Dato confidencial – acceso solo umu

BADAR TAGMOUTI ABDOUNE

Dato confidencial – acceso solo umu

PABLO MANRESA SOLER

**Grupo 1
Subgrupo 3**



ÍNDICE

INTRODUCCIÓN	2
LENGUAJE MINIC	2
『 TOKENS 』	2
『 FLEX 』	3
『 BISON 』	3
ANÁLISIS SINTÁCTICO	3
GENERACIÓN DE CODIGO MIPS.....	6
♦ FUNCIONES PARA GENERACIÓN DE CÓDIGO	6
♦ IMPLEMENTACIÓN LISTACODIGO	8
♦ GENERACIÓN DE CÓDIGO I	9
♦ GENERACIÓN DE CÓDIGO II	10
♦ EJEMPLOS DE CODIGO	15
CONCLUSIÓN	17



INTRODUCCIÓN

El proyecto “MiniC” es una práctica de Compiladores en la que consiste en crear un mini compilador con los conocimientos adquiridos en la teoría y habilidades creadas en la práctica.

A continuación, se explicará todo el procedimiento que se ha ido realizando para poder llevar a cabo la práctica.

LENGUAJE MINIC

MiniC es basa en la estructura del lenguaje C, pero a una escala simplificada. No tenemos operadores lógicos, solo manejamos constantes y variables enteras consiguiendo así representarlos de forma booleana, el valor 0 es false, y cualquier otro numero true.

Tenemos sentencias, *if*, *if-else*, *while*, *read*, *print*, *expresiones aritméticas*.

『 TOKENS 』

En el fichero *minic.h* podemos visualizar todos los tokens (void, var, const...) con los que iremos trabajando a la hora de realizar la práctica.

De dichos tokens podemos destacar algunas cualidades, enteros **{entero}** debe estar entre el -2^{31} y 2^{31} , identificadores **{letra}|_({letra})|{digito}|_*** son una secuencia de letra, dígitos y símbolo subrayado, aparte solo deben tener como máximo 16 caracteres y no comenzar con dígito.

```
#define VOID 1
#define PRINT 2
#define READ 3
#define VAR 4
#define CONST 5
#define IF 6
#define ELSE 7
#define WHILE 8
#define ID 9
#define INT 10
#define STRING 11
#define LLAVEI 12
#define LLAVED 13
#define PARENI 14
#define PAREND 15
#define PUNTCOM 16
#define COMMA 17
#define ASSIGNOP 18
#define PLUSOP 19
#define MINUSOP 20
#define DIVOP 21
#define MULTOP 22
```



『 FLEX 』

Para diseñar nuestro analizador léxico. Hemos tenido que utilizar la herramienta flex, herramienta creada para generar analizadores léxicos. Todo lo relacionado con Flex lo podemos encontrar en el fichero *minic.l*

Lo primero de todo, se tienen que identificar los tokens asignándole a cada token una expresión regular, pudiendo así identificarlos.

Para entero, hemos creado la función *int_function()* que comprueba si el entero se encuentra entre el -27^1 y 27^1 . *Id_function()* comprueba si el identificador no supera los 16 caracteres, entre otras funciones para mostrar los errores, léxicos y comentarios multilínea, consiguiendo así avisar al usuario.

『 BISON 』

ANÁLISIS SINTÁCTICO

En la parte del análisis sintáctico, la función que se desempeña es comprobar el identificador pertenece a la tabla símbolos, en caso de que no pertenezca nos encontramos frente ciertos errores, como que la variable no esta declarada, la variable esta redeclarada, asignación de un valor a una constante.

En este mismo apartado, creamos una variable tipo, con la finalidad de almacenar que tipo de parámetro estamos guardando, si tipo VARIABLE o tipo CONSTANTE, con la finalidad de que a la hora de llegar hacer la producción de asignación comprobamos si esta asignando a una constante, cosa que no se puede hacer dando lugar a un error.

Para ello se han creado ciertas funciones, en este apartado.

```
bool perteneceTablaS(char *lexema){  
  
    PosicionLista pos = buscaLS(tablaSimbolos,lexema);  
    if (pos != finallS(tablaSimbolos)) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

perteneceTablaS(), es una función booleana, que comprueba el lexema que se le pasa por parámetro si pertenece a la tabla. Consiguiendo buscar en la *tablaSimbolos* el lexema y guardando la posición, si dicha posición es distinta de la última eso quiere decir que tenemos el lexema en la tabla por lo tanto devolvemos true, en caso contrario devolvemos false.



```
void anyadeEntrada(char *lexema, Tipo tipo, int valorContenido){  
  
    Simbolo aux;  
    aux.nombre = lexema;  
    aux.tipo = tipo;  
    aux.valor = valorContenido;  
  
    insertaLS(tablaSimbolos, finalLS(tablaSimbolos), aux);  
  
}
```

anyadeEntrada(), es un procedimiento que se le llama cuando la función mencionada anteriormente nos devuelve false, interpretando así que no está en la tablaSimbolos y por lo tanto hay que añadirlo.

Este procedimiento, recibe como parámetro tres valores, el lexema, un tipo (enumerado), el valorContenido. Con esos parámetros creamos un símbolo y lo insertamos en posición final de la tablaSimbolos. Hay que recalcar que este procedimiento también lo utilizaremos más tarde para los strings, por lo tanto, la diferencia entre los strings y las variables o constantes está en el valorContenido, aparte del tipo. Las variables y constantes tendrán un -1, y los strings tendrán valor de un contador que utilizaremos para poder crear una etiqueta con dicho número en función de la cantidad de strings que tuviésemos, un ejemplo sería \$str1, el 1 el valor que tendríamos en el campo valor símbolo del string.

```
bool esConstante(char *lexema){  
  
    PosicionLista pos = buscaLS(tablaSimbolos, lexema);  
    if(pos != finalLS(tablaSimbolos)){  
        Simbolo aux = recuperaLS(tablaSimbolos, pos);  
  
        if (aux.tipo == CONSTANTE){  
            return true;  
        } else {  
            return false;  
        }  
    } else {  
        /* No está en la tabla, debido a que la posición es igual que la final */  
        return false;  
    }  
}
```

esConstante(), como su nombre dice, comprueba si el lexema que le pasamos por parámetro es una constante o no. El funcionamiento es obtener la posición de dicho lexema de la tablaSimbolos. Y comprobar si es o no es la posición final, en caso de no serlo recuperar el símbolo que hay en esa posición y comprobar el campo tipo.



```

void imprimirTablaS() {
    /* Imprimimos la tabla */
    printf("#####\n");
    printf(".data\n");

    Lista listaID = creaLS(); // lista de ID (variables o constantes)
    Lista listaCAD = creaLS(); // lista de ID (cadenas)

    PosicionLista p = inicioLS(tablaSimbolos);
    PosicionLista final = finalLS(tablaSimbolos);

    /* Recorremos la lista de simbolos principal y en funcion de tipo lo asignamos a una de las listas locales */
    while(p != final) {
        Simbolo aux = recuperaLS(tablaSimbolos,p);

        /* En caso de ser
            variable o constante -> "_nombreVariable"
            cadena -> "$strNumero" */

        switch(aux.tipo){
            case VARIABLE:
                insertaLS(listaID,finalLS(listaID),aux);
                break;
            case CONSTANTE:
                insertaLS(listaID,finalLS(listaID),aux);
                break;
            case CADENA:
                insertaLS(listaCAD,finalLS(listaCAD),aux);
                break;
            default:
                break;
        }

        p = siguienteLS(tablaSimbolos,p);
    }

    /* Despues de guardar en las listas segun el simbolo procedemos a imprimirlas */
    /* LA FINALIDAD DE ESTO ES QUE SE MUESTRE PRIMERO LOS STRINGS Y DESPUES LOS IDENTIFIERS */

    /* MOSTRAMOS LOS STRINGS */
    printf("# STRINGS #####\n");

    PosicionLista posCad = inicioLS(listaCAD);
    PosicionLista finalCad = finalLS(listaCAD);

    while(posCad != finalCad){
        Simbolo auxC = recuperaLS(listaCAD,posCad);
        printf("$str%d: .asciz %s\n",auxC.valor,auxC.nombre);

        posCad = siguienteLS(listaCAD,posCad);
    }
    printf("\n");

    /* MOSTRAMOS LOS IDENTIFIERS */
    printf("# IDENTIFIERS #####\n");
    PosicionLista posID = inicioLS(listaID);
    PosicionLista finalID = finalLS(listaID);

    while(posID != finalID){
        Simbolo auxID = recuperaLS(listaID,posID);
        printf("_%s: .word 0\n",auxID.nombre);

        posID = siguienteLS(listaID,posID);
    }

    /* Liberamos las listas locales creadas */
    liberaLS(listaID);
    liberaLS(listaCAD);
}

```

imprimirTablaS(), es un procedimiento que al final utilizaremos para después mostrar la parte “.data” de ensamblador en la que se guardan los strings y las variables/constantes. En dicho procedimiento creamos dos Listas auxiliares, una para guardar los strings y otra para guardar las variables y las constantes, recorriendo así la lista principal y clasificarlos en función del tipo. Para que más tarde proceder a mostrarlos.

Hay que recalcar de que se ha hecho uso de la librería *listaSimbolo.c* y *listaSimbolo.h* que proporcionan los profesores de la materia.



GENERACIÓN DE CODIGO MIPS

♦ FUNCIONES PARA GENERACIÓN DE CÓDIGO

Una vez realizado el análisis sintáctico, podemos generar el código de las expresiones, de los identificadores y los enteros.

Comenzando primero por las funciones que necesitaremos en dicho apartado

```
char *obtenerRegistro(){  
  
    int posicion;  
    for (int i = 0; i < MAX_REG; i++){  
        if(registro[i] == 0){  
            posicion = i;  
            registro[i] = 1;  
            break;  
        }  
    }  
  
    char temporal[32];  
    sprintf(temporal, "%t%d", posicion);  
    return strdup(temporal);  
}
```

obtenerRegistro(), función que recorre el array registro (variable global) y nos devuelve la primera posición que tenga ese array libre, es decir su contenido a 0. Ponemos dicho código 1, debido a que lo vamos a utilizar. Mediante la función *sprintf()* guardamos en el array temporal, según el formato que tenemos puesto. Y lo devolvemos.

```
void liberarRegistro(char *reg){  
  
    for (int i = 0; i < MAX_REG; i++){  
        if (arrayTempAux[i] == reg[2]){  
            // Procedemos a liberar la posicion en el array de registros booleanos  
            registro[i] = 0;  
        }  
    }  
}
```

liberarRegistro(), esta función es necesaria debido a que si nos fijamos en ensamblador los registros temporales son desde 0 a 9, por lo tanto una vez ocupados todos tendremos que ir liberándolos para poder seguir utilizándolos.

liberarRegistro recibe como parámetro reg, que contendrá \$t y un numero concatenado (ejemplo \$t5), por lo tanto nosotros recorreremos el arrayTemAux (variable global) y comprobamos la segunda posición de reg, es decir el numero para que podamos acceder a nuestro array de registros y lo pongamos a cero, está libre.



```
char *concatenar(char *nombre){  
  
    char temporal[32];  
    sprintf(temporal, "_%s", nombre);  
    return strdup(temporal);  
  
}
```

concatener(), el uso de dicha función es como su nombre dice concatena el nombre del identificador (recibimos por parámetro) mediante el formato “_nombre” (ejemplo _b) para ello utiliza la misma función que habíamos mencionado en la función *obtenerRegistro()*.

```
void imprimirTablaC(){  
    printf("\n# MAIN #####\n");  
    printf("\t.text\n");  
    printf("\n\t.globl main\n\n");  
    printf("main:\n");  
  
    PosicionListaC p = inicioLC(tablaCodigo);  
    while (p != finalLC(tablaCodigo)) {  
        Operacion oper;  
        oper = recuperaLC(tablaCodigo,p);  
        printf("%s",oper.op);  
        if (oper.res) printf("\t%s",oper.res);  
        if (oper.arg1) printf(",%s",oper.arg1);  
        if (oper.arg2) printf(",%s",oper.arg2);  
        printf("\n");  
        p = siguienteLC(tablaCodigo,p);  
    }  
  
    // Instruccion de que hemos terminado  
    printf("\n\tli \t$v0, 10\n");  
    printf("\t\tsyscall\n");  
  
}
```

Por último, imprimirTablaC(), es un procedimiento que utilizaremos para mostrar todo el código.

◆ IMPLEMENTACIÓN LISTACODIGO

Hay que recalcar de que se ha hecho uso de la librería *listaCodigo.c* y *listaCodigo.h* que proporcionan los profesores de la materia, pero con cierta modificación.

```
struct PosicionListaCRep {
    Operacion dato;
    struct PosicionListaCRep *sig; // apunta al siguiente
    struct PosicionListaCRep *atr; // apunta al anterior
};
```

PosicionListaRep, tiene tres campos: Operación dato, Posición *sig y Posición *atr. El motivo de dicha modificación es para poder retroceder en la lista y no solo avanzar, consiguiendo así una forma más fácil de general el código.

```
ListaC creaLC() {
    ListaC nueva = malloc(sizeof(struct ListaCRep));
    nueva->cabecera = malloc(sizeof(struct PosicionListaCRep));
    nueva->cabecera->sig = NULL;
    nueva->cabecera->atr = NULL;
    nueva->ultimo = nueva->cabecera;
    nueva->n = 0;
    nueva->res = NULL;
    return nueva;
}
```

Respecto a la Lista creaLC() que ofrecen los profesores de la materia, la nueva función se le añade que el "nueva->cabecera->atr = NULL"

```
void insertaLC(ListaC codigo, PosicionListaC p, Operacion o) {
    NodoPtr nuevo = malloc(sizeof(struct PosicionListaCRep));
    nuevo->dato = o;
    nuevo->sig = p->sig;
    nuevo->atr = p;
    p->sig = nuevo;

    if (codigo->ultimo == p) {
        codigo->ultimo = nuevo;
    } else {
        nuevo->sig->atr = nuevo;
    }
    (codigo->n)++;
}
```

En insertaLC() añadimos que "nuevo->atr = p" y que en caso de que no sea el ultimo que "nuevo->sig->atr = nuevo"

```
PosicionListaC anteriorLC(ListaC codigo, PosicionListaC p) {
    return p->atr;
}
```

anteriorLC() nueva función, que nos devuelve la posición anterior en la lista

♦ GENERACIÓN DE CÓDIGO I

Una vez mencionado todas las modificaciones que se han hecho y las funciones que se han creado para la generación de código procedemos con la primera parte de la generación de código de MIPS.

[para más explicación, se encuentra el código *minic.y*]

Para los enteros INT e identificadores ID, creamos una operación según la instrucción que le corresponda

INT	ID
Li	Lw
<code>obtenerRegistro()</code>	<code>obtenerRegistro()</code>
\$1	<code>concatenar(\$1)</code>
NULL	NULL

Para las expresiones, lo que hacemos primero es hacer que apunte \$\$ a \$1, concatenamos a \$\$ \$3 y después lo liberamos. Debido a que el funcionamiento de `concatneLC()` es copiar el segundo parámetro en el primero, haciendo así que tuviésemos un duplicado por lo tanto deberíamos liberarlo, y creamos la operación en función del token, si es PLUSOP seria add, si es MINUSOP seria sub, MULTOP seria mult, DIVOP seria div. Insertamos la operación en la `tablaCodigo`, guardamos el resultado de \$\$ en `oper.res` y liberamos el registro que habíamos reservado.

EXPRESSION			
ADD	SUB	MULT	DIV
<code>recuperaResLC(\$1)</code>			
<code>recuperaResLC(\$1)</code>			
<code>recuperaResLC(\$3)</code>			

Para la cláusula de negación, minus, creamos una operación con la instrucción adecuada a ella y guardamos en el registro que teníamos el dato.



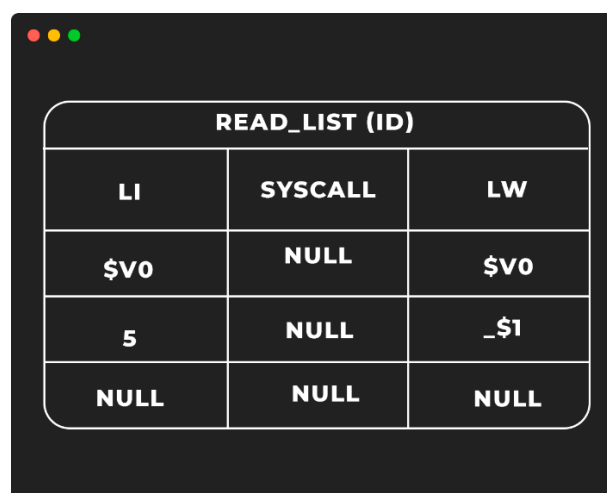
Y, por último, para la clausula del paréntesis no hacemos nada, solo le pasamos al padre el contenido que tiene en \$2.

◆ GENERACIÓN DE CÓDIGO II

Una vez terminado con la generación de código de las expresiones aritméticas, pasamos para las sentencias.

[para más explicación, se encuentra el código *minic.y*]

Comenzamos por:



The diagram shows a window titled **READ_LIST (ID)** containing a table with four rows and three columns:

READ_LIST (ID)		
LI	SYSCALL	LW
\$V0	NULL	\$V0
5	NULL	_\$1
NULL	NULL	NULL

Cada vez que vamos creando la instrucción, la vamos insertando en la última posición de la tablaCodigo.

Para el caso de print_item.

PRINT_ITEM (STRING)		
LW	LI	SYSCALL
\$A0	\$V0	NULL
SstrContadorCadenas	4	NULL
NULL	NULL	NULL

PRINT_ITEM (EXPRESSION)		
MOVE	LI	SYSCALL
\$A0	\$V0	NULL
recuperaResLC(\$1)	1	NULL
NULL	NULL	NULL

Sin embargo, para print_list, no es necesario crear ninguna operación. Para cuando desemboque en print_item le pasaremos el valor \$1 a \$\$ y para cuando desemboque en print_list COMMA print_item, haremos algo parecido. \$1 se asigna a \$\$ y a \$\$ le concatenamos \$3 pero después liberando \$3.

Ahora una vez llegado a statement nos encontramos con varias clausuras.

READ read_list PUNTCOM, en el que le asignamos a \$\$ \$2 y haremos lo mismo en PRINT print_list PUNTCOM.

Para el caso de while, lo que hacemos es guardar antes de statemen la posición final de la tablaCodigo para después poder insertar correctamente las operaciones.

WHILE			
ETIQUETA	BEQZ	B	ETIQUETA2
NULL	recuperaResLC(\$3)	ETIQUETA	NULL
NULL	ETIQUETA2	NULL	NULL
NULL	NULL	NULL	NULL

En la sentencia if, lo primero que hacemos es contar cuantas instrucciones tiene el statement, para que después podamos retroceder esa cantidad y poner el beqz en la posición correcta.



If-else, realizamos lo mismo que en el if simple, contamos cuantas instrucciones tiene el primer statement para poder retroceder esa cantidad de posiciones en la lista e insertar el beqz en la posición correcta.



Para la última clausura de statement, tenemos la asignación a un id.



Una vez terminado con todas las clausuras de statement, pasamos a `statement_list`. Que tiene lambda, que sería crear una lista vacía y asignársela al padre `$$`. Y en la otra clausura, asignamos a `$$ $1`, y le concatenamos a `$$ $2` y liberamos `$2`.

Ahora que hemos terminado con el bloque de sentencias procedemos a seguir con el bloque de declaraciones para que podamos ejecutar el código correctamente.

En `asig`, tenemos ID, básicamente si no pertenece a la tabla de símbolos, lo que hacemos es añadirlo y crear una lista vacía para asignársela al padre. Sin embargo, para ID `ASSIGNOP` expresión, realizamos la misma comprobación. Si no pertenece lo añadimos y creamos la operación adecuada.



Para `identifier_list`, realizamos el mismo procedimiento que en el bloque de sentencias para `statement_list`. En `asig` le asignamos `$1` a `$$` y en `identifier_list COMMA asig`, a `$$` le asignamos `$1`, concatenamos `$3` a `$$` y liberamos `$3`.



Antes de llegar a la última clausura `program`. Nos encontramos con `declarations`, que básicamente se divide en tres clausuras, una vacía, una en la que tenemos el token `CONST` y otra en la que tenemos el token `VAR`.

En la vacía, a `$$` le insertamos una lista vacía, para las otras dos clausuras a `$$` le asignamos `$1`, concatenamos `$3` a `$$` y liberamos `$3`. Pero en tipo guardamos datos diferentes. Si es `CONST` guardamos `CONSTANTE` y si es `VAR` guardamos `VARIABLE`.

Una vez llegado a `program`, encontramos que al principio creamos las tablas (`tablaSimbolo`, `tablaCodigo`) y al final las mostramos y liberamos.



◆ EJEMPLOS DE CODIGO

```

/*****
 * Fichero de prueba nº 1
 *****/

void prueba() {
    const a =0, b=0;
    var c = 10+2-2;
    print "Inicio del programa\n";

    if(a) print "Hola";

    while(c){
        while(a){
            print "na";
        }
    }
}

```

```

#####
.data
# STRINGS #####
$str1: .asciiz "Inicio del programa\n"
$str2: .asciiz "Hola"
$str3: .asciiz "na"

# IDENTIFIERS #####
_a: .word 0
_b: .word 0
_c: .word 0

# MAIN #####
.text

.globl main

main:
    li $t0,0
    sw $t0,_a
    li $t1,0
    sw $t1,_b
    li $t2,10
    li $t3,2
    add $t2,$t2,$t3
    li $t3,2
    sub $t2,$t2,$t3
    sw $t2,_c

    lw $a0,$str1
    li $v0,4
    syscall

    lw $t3,_a
    beqz $t4,$l1

    lw $a0,$str2
    li $v0,4
    syscall
$l1:

    lw $t5,_c
    lw $t6,_a
$l4:
    beqz $t5,$l5
$l2:
    beqz $t6,$l3

    lw $a0,$str3
    li $v0,4
    syscall
    b $l2
$l3:
    b $l4
$l5:

    li $v0, 10
    syscall

```




```

/*****
* Fichero de prueba nº 1
*****/

void prueba() {

    const b = 6 * 2;
    var c;
    var d = 6 + 1, e = 10 / 2;

    print "Asignatura de Compiladores\n";

    print "Introduce el valor de \"c\":\n";
    read c;

    if (c) print "\"c\" no era nulo.", "\n";
        else print "\"c\" si era nulo.", "\n";
}

```

```

#####
.data
# STRINGS #####
$str1: .asciiz "Asignatura de Compiladores\n"
$str2: .asciiz "Introduce el valor de \"c\":\n"
$str3: .asciiz "\"c\" no era nulo."
$str4: .asciiz "\n"
$str5: .asciiz "\"c\" si era nulo."
$str6: .asciiz "\n"

# IDENTIFIERS #####
_b: .word 0
_c: .word 0
_d: .word 0
_e: .word 0

# MAIN #####
.text

.globl main

main:
    li $t0,6
    li $t1,2
    mul $t0,$t0,$t1
    sw $t0,_b
    li $t1,6
    li $t2,1
    add $t1,$t1,$t2
    sw $t1,_d
    li $t2,10
    li $t3,2
    div $t2,$t2,$t3
    sw $t2,_e

    lw $a0,$str1
    li $v0,4
    syscall

    lw $a0,$str2
    li $v0,4
    syscall

    li $v0,5
    syscall
    sw $v0,_c

    lw $t3,_c
    beqz $t3,$l1

    lw $a0,$str3
    li $v0,4
    syscall

    lw $a0,$str4
    li $v0,4
    syscall
    b $l2
$l1:

    lw $a0,$str5
    li $v0,4
    syscall

    lw $a0,$str6
    li $v0,4
    syscall
$l2:

    li $v0, 10
    syscall

```



CONCLUSIÓN

Como conclusión, la realización de esta tarea al principio, en concreto los primeros días nos ha resultado un poco complicada debido a la cantidad de conocimiento e información que debíamos procesar. Pero poco a poco la hemos conseguido realizar, con trabajo y dedicación, y ayuda del profesorado.

A nuestro parecer, el realizar esta práctica nos ha ayudado bastante a entender el concepto de mini compilador y como funcionan los traductores, aunque sea a un modo muy reducido.