|  |  |
| --- | --- |
|  | **2015** |
|  | Vilella Trigueros, Alejandro  alejandro.vilella@um.es  Grupo: 3.3  Ballesta Garres, Pedro  [pedro.ballesta@um.es](mailto:pedro.ballesta@um.es)  Grupo: 3.3  Profesor: María Antonia Martínez Carreras  Convocatoria: Julio 2015 |

|  |
| --- |
| **[Compilador minic]** |
| Se consigue realizar un programa único que pueda leer un programa en miniC y genere otro equivalente en código MIPS. Podremos ejecutar este programa resultante usando Spin o Mars, a modo de intérprete del ensamblador de MIPS. |

ÍNDICE

[Introducción 3](#_Toc424025497)

[Símbolos terminales 3](#_Toc424025498)

[Análisis Léxico 3](#_Toc424025499)

[Análisis sintáctico 4](#_Toc424025500)

[Generación de código MIPS 5](#_Toc424025501)

[Manual de usuario para la ejecución 7](#_Toc424025502)

[Pruebas de ejecución 7](#_Toc424025503)

# Introducción

Partiremos de un lenguaje al que llamaremos miniC, muy simplificado, donde luego podremos ir ampliando su sintaxis opcionalmente. El compilador tendrá una serie de restricciones.

* Sólo manejará constantes y variables enteras.
* Los valores verdadero y falso, como resultado de expresiones condicionales, se interpretarán como en C, falso será un 0 y todo lo demás verdadero.
* Los operadores relacionales y lógicos desaparecen.
* Las sentencias de flujo de control se reducirán a if, if-else y while.

## Símbolos terminales

* Constantes:
  + Enteros: Desde -231 hasta 231.
  + Cadenas: Secuencia de caracteres delimitados por comillas dobles.
* Identificadores: Secuencia de letras, dígitos y símbolos de subrayado, no comenzando por dígito y no excediendo los 16 caracteres.
* Palabras reservadas: func, var, let, if, else, while, print y read.
* Caracteres especiales: **;**, **,**, **+**, **-**, **\***, **/**, **=**, **(**, **)**, **{**, **}**.

# Análisis Léxico

El analizador léxico es la primera fase de un compilador y es un fichero donde se implementan los tokens y lo que devuelve cada token. Un token es un par formado por un nombre de token y un atributo opcional.

El formato de un analizador léxico es el siguente:

{definiciones}

%%

{reglas}

%%

{subrutinas de usuario}

* Definiciones: Se especifican macros que dan nombre a expresiones regulares auxiliares que serán utilizadas en la sección de reglas.
* Reglas: En esta sección hay secuencias de expresiones regulares y una acción a cada expresión regular. Esto es un bloque de sentencias en C que se ejecutan cuando se reconoce el lexema asociado a esa expresión regular.
* Subrutinas de usuarios: Precedimiento auxiliares necesarios. (Nosotros en el proyecto no hemos generado ninguna subrutina).

Nuestra forma de analizar los errores es a través del modo pánico. La función del modo pánico es omitir los errores léxicos dentro de una palabra reservada.

Los ficheros asociados al analizador léxico son:

* fichero.l: Es un fichero que contiene la especificación del análisis léxico explicada su estructura arriba y que se procesa con el progrma Flex.
* Semántico.tab.h: Es un fichero que crea Flex asociado al fichero que contiene la especificación del análisis léxico.

# Análisis sintáctico

El objetivo del analizador sintáctico es el de producir, a partir de una cadena de componentes léxicos, una salida que consistirá en:

* Un árbol sintáctico que se proporcionará a la siguiente etapa del front-end, en el caso de que la secuencia de tokens verifique la gramática libre de contexto del lenguaje fuente.
* Si la secuencia de tokens contiene errores sintácticos, es decir, alguna frase no se ajusta a la estructura sintáctica definida por la gramática del lenguaje fuente, el compilador generará un informe con la lista de errores detectados. Este informe deberá ser lo más claro y exacto posible.

Las principales funciones que hace el analizador sintáctico son:

* Aceptar lo que es válido sintácticamente y rechazar lo que no lo es.
* Hacer explícito el orden jerárquico que tienen los operadores en el lenguaje.
* Guiar el proceso de traducción dirigida por la sintaxis.
* Acceder a la tabla de símbolos.
* Chequeo de tipos (del analizador semántico).
* Generar código intermedio.
* Generar errores cuando se producen.

# Generación de código MIPS

La generación de código es la más díficil y es la última parte del compilador, pasar todo el código de C a código ensamblador.

Voy a ir explicando las funciones que hemos utilizado para la generación de código.

* El fichero codigo.c contiene funciones auxiliares para la ayuda en el fichero miniC.y a la generación de código. Algunas de estas funciones son:
  + typedef struct linea{}: Es una estrutura de datos para generar lineas en código ensamblador. Contiene la operación y 3 argumentos y el registro que este usa.
  + typedef struct codigo{}: Es otra estructura que se usará para concatenar líneas . Contiene las líneas de código generado en el orden que deben de ser ejecutadas.
  + Linea \*crearLinea(): Reserva espacio en memoria y devuelve una línea de código MIPS con la operación y los tres argumentos.
  + Char \*registroLibre(): Busca si hay un registro libre para poder usarlo, en caso de que no hayan mostrará un mensaje de registros agotados.
  + Void inicializaPadre(): Inicializa los campos del padre para que no salga error(NULL POINTED EXCEPTION).
  + Void concatenaLineaACodigo(): Concatena una línea de con el código del padre, siempre por detrás.
  + Void comprobarIdentificador(): Comprueba si existe la variable en la tabla de símbolos.
  + Char \*concatenarString(): Devuelve la concatenación de un prefijo y un sufijo(la usaremos para poner un \_).
  + Void concatenarCodigo(): Concatena dos códigos que se le pasan por parámetro.
  + Int nulo(): Si el código que se pasa por parámetro es nulo devuelve 1 en otro caso devuelve 0.
  + Void liberarRegistros(): Libera todos los registros.
  + Char \*crearTag(): Crea etiquetas para el código MIPS.
  + Void imprimir(): imprime el código MIPS a un fichero de salida.
* El fichero id.c contiene funciones auxiliares para manejar la tabla de símbolos. Estas funciones son:
  + Typedef struct identificador{}: Es una estructura de datos para las variables. Contiene el nombre, valor y el tipo.
  + Void inicializacionTablaSimbolos(): Inicializa a null la tabla de símbolos.
  + Void establerTipo(): Establece el tipo del identificador.
  + Void insertarID(): Si el identificador no existe en la tabla de símbolos lo guarda.
  + Void existeID(): Busca en la tabla de símbolos para ver si existe la variable.
  + Void insertarTablaSimbolos(): Si la variable no está en la tabla de símbolos la añade.
  + Int existeEnTablaSimbolos(): Devuelve un 0 si no existe la variable en la tabla de símbolos.
  + Identificador \*buscaID(): Devuelve el identificador de busca, si no lo encuentra devuelve NULL.
  + Int esConstante(): Devulve si el identificador es una variable o una constante.
  + Void imprimirID(): Imprime la cabecera del código ensamblador, es decir, la parte de las variables globales, en un fichero de salida.
  + Void borrarTablaSimbolos(): Borra toda la tabla de símbolos.
* El fichero string.c contiene funciones auxiliares para manejar cadenas. Estas funciones son:
  + typedef struct tString{}: Es una estructura de datos para manejar las cadenas. Contiene una cadena y nombreGenerico.
  + Void inicializarString(): Inicializa las cadenas a NULL.
  + Void insertarString():
  + Int buscarNumeroCadena(): Devulve el número de cadenas.
  + Char \*buscarString(): Devulve el nombre de la cadena, si no existe devuelve NULL.
  + Void imprimirString(): Imprime en el fichero de MIPS las cadenas que hay en el programa.
  + Void borrarString(): Borra todas las cadenas.
* El fichero miniC.y que es donde se genera todo el código MIPS y donde se usan las funciones auxiliares para la creación de dicho código. En este fichero tenemos otra función:
  + Void yyerror(): Muestra por pantalla el error que se ha producido y en que línea ha sido.
* El fichero main.c es donde contiene la función para imprimir todo el código MIPS a un fichero de salida.
* El fichero makefile: Se utiliza para compilar todo el proyecto y crear el compilador.

# Manual de usuario para la ejecución

Para que pueda funcionar y ejecutar el programa se necesita una clase de aplicaciones:

* El analizador léxico Flex.
* El analizador sintáctico Bison.
* El GCC para compilar el código C.
* Un programa para lenguaje ensamblador como MIPS.
* El sistema operativo Linux.
* Y un terminal para ejecutar el programa.

Para ejecutar la aplicación se hace en una serie de pasos:

1. Abrir un terminal en linux.
2. Ir al directorio donde se encuentra el proyecto.
3. Ejecutar make.
4. Y ejecutar ./compiladorMiniC fichero\_entrada
5. Pasar el código resultante a MARS o similar para ejecutar.

# Pruebas de ejecución

**Prueba 1:**

Entrada:

func prueba () {

let a=0, b=0;

var c=5+2-2;

print "Inicio del programa\n";

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";

}

Código ensamblador:

.data

# Cadenas del programa

$str1:

.asciiz "Inicio del programa\n"

$str2:

.asciiz "a"

$str3:

.asciiz "\n"

$str4:

.asciiz "No a y b\n"

$str5:

.asciiz "c = "

$str6:

.asciiz "Final"

# Variables globales

\_a:

.word 0

\_b:

.word 0

\_c:

.word 0

.text

main:

li $t0 0

sw $t0 \_a

li $t0 0

sw $t0 \_b

li $t0 5

li $t1 2

add $t0 $t0 $t1

li $t1 2

sub $t0 $t0 $t1

sw $t0 \_c

li $v0 4

la $a0 $str1

syscall

lw $t0 \_a

beqz $t0 $l5

li $v0 4

la $a0 $str2

syscall

li $v0 4

la $a0 $str3

syscall

b $l6

$l5:

lw $t1 \_b

beqz $t1 $l3

li $v0 4

la $a0 $str4

syscall

b $l4

$l3:

$l1:

lw $t2 \_c

beqz $t2 $l2

li $v0 4

la $a0 $str5

syscall

lw $t3 \_c

move $a0 $t3

li $v0 1

syscall

li $v0 4

la $a0 $str3

syscall

lw $t3 \_c

li $t4 2

sub $t3 $t3 $t4

li $t4 1

add $t3 $t3 $t4

sw $t3 \_c

b $l1

$l2:

$l4:

$l6:

li $v0 4

la $a0 $str6

syscall

li $v0 4

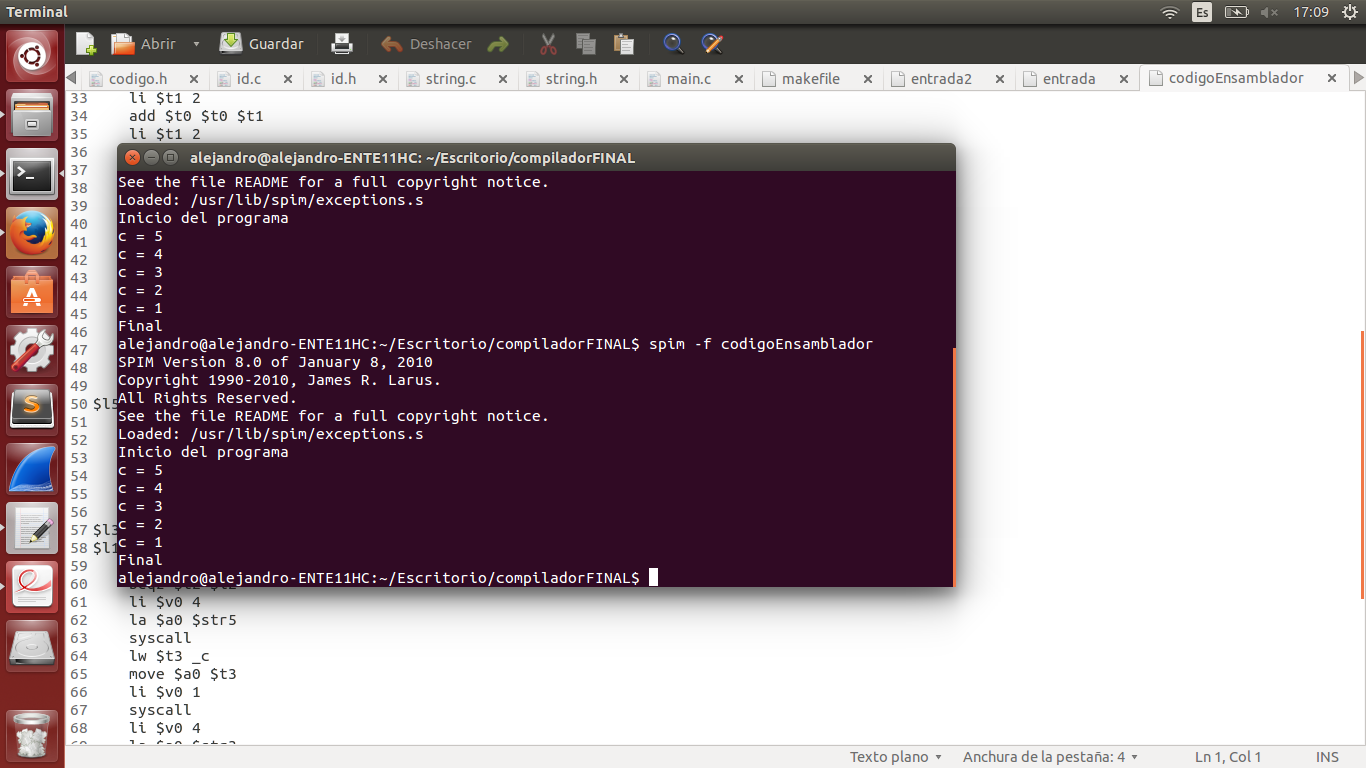
la $a0 $str3

syscall

li $v0 10

syscall

Captura en ejecución:



Prueba 2:

Entrada:

func prueba () {

let a=0, b=0;

var c=5+2-2,i;

let d=3;

var j;

print "Inicio del programa\n";

read c;

read j;

while (c)

{

print "c = ",c,"\n";

c = c-2+1;

}

print "j","\n";

print "Final","\n";

}

Código ensamblador:

.data

# Cadenas del programa

$str1:

.asciiz "Inicio del programa\n"

$str2:

.asciiz "c = "

$str3:

.asciiz "\n"

$str4:

.asciiz "j"

$str5:

.asciiz "Final"

# Variables globales

\_a:

.word 0

\_b:

.word 0

\_c:

.word 0

\_i:

.word 0

\_d:

.word 0

\_j:

.word 0

.text

main:

li $t0 0

sw $t0 \_a

li $t0 0

sw $t0 \_b

li $t0 5

li $t1 2

add $t0 $t0 $t1

li $t1 2

sub $t0 $t0 $t1

sw $t0 \_c

li $t0 3

sw $t0 \_d

li $v0 4

la $a0 $str1

syscall

li $v0 5

syscall

sw $v0 \_c

li $v0 5

syscall

sw $v0 \_j

$l1:

lw $t0 \_c

beqz $t0 $l2

li $v0 4

la $a0 $str2

syscall

lw $t1 \_c

move $a0 $t1

li $v0 1

syscall

li $v0 4

la $a0 $str3

syscall

lw $t1 \_c

li $t2 2

sub $t1 $t1 $t2

li $t2 1

add $t1 $t1 $t2

sw $t1 \_c

b $l1

$l2:

li $v0 4

la $a0 $str4

syscall

li $v0 4

la $a0 $str3

syscall

li $v0 4

la $a0 $str5

syscall

li $v0 4

la $a0 $str3

syscall

li $v0 10

syscall

Captura:

