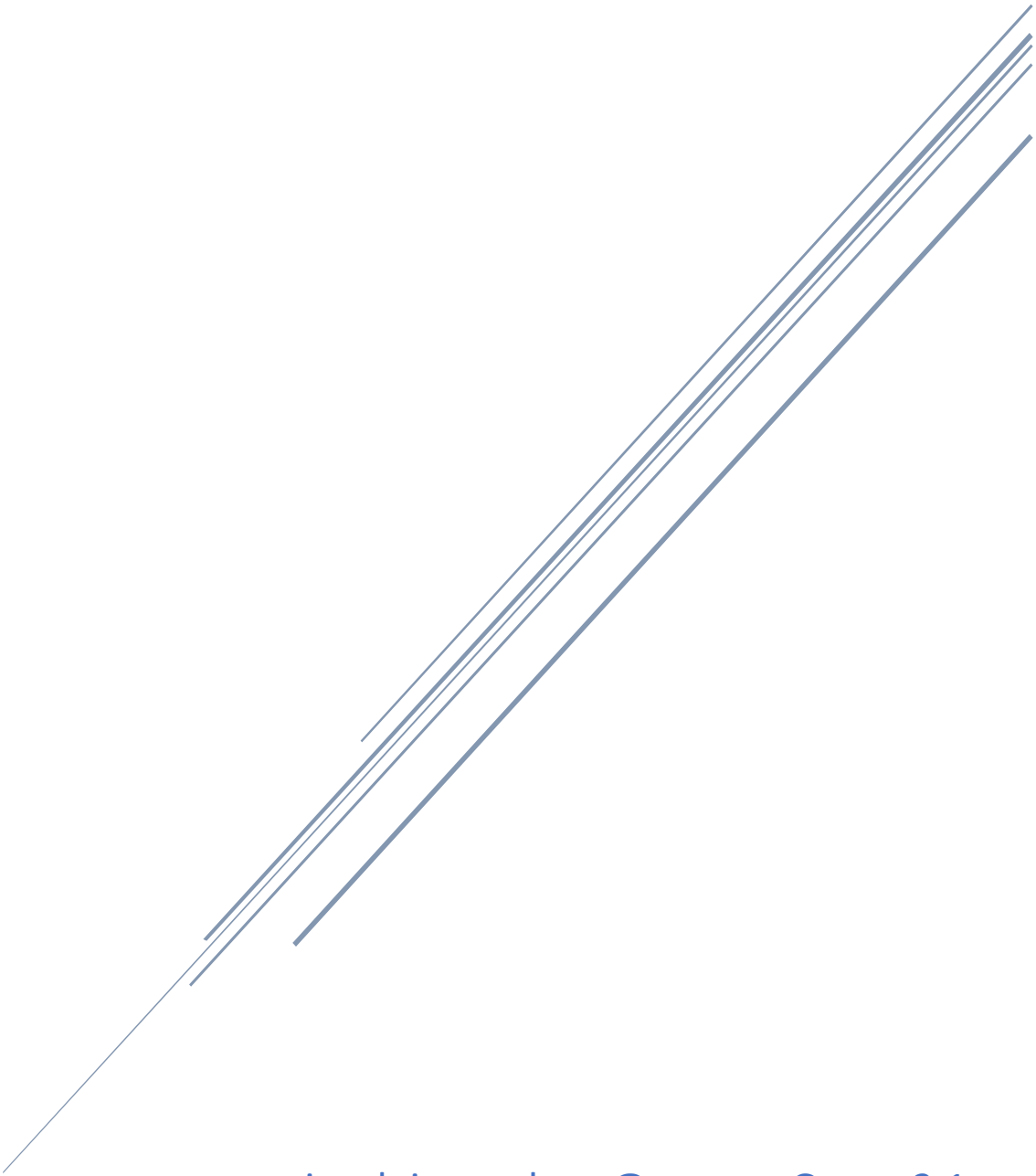
PRÁCTICAS COMPILADORES MINIC JUNIO 2021

José Luis Sánchez Carrasco

Eduardo Espinosa Meroño



joseluis.sanchezc@um.es - Grupo 3.1
eduardo.espinosam@um.es – Grupo 3.1

ÍNDICE

ANÁLISIS LÉXICO

El analizador léxico **lexico.l** está compuesto por 2 secciones, en la primera de ellas se encarga de definir mediante expresiones regulares los tipos de datos que va a manejar el compilador, en este caso define lo que es un **digito**, un **carácter**, un **entero** y además define **pánico**, que mediante una expresión regular detecta todo carácter no válido en el vocabulario del lenguaje, esto se utiliza más adelante para detectar errores léxicos. También se define la opción **yylineno** para devolver la línea en la que se encuentra un carácter y **comentario** para manejar los comentarios encapsulados que se explican más adelante.

Como segunda sección, el analizador léxico define los tokens que va a manejar el compilador, destacamos la definición de los espacios en blanco con ***[\n\t]+*** , la definición de los comentarios que se pueden hacer en miniC, en primer lugar se puede hacer un comentario simple utilizando **//** , lo cual se define con la siguiente expresión regular: ***"//"(.*)[\n]*** y en segundo lugar se puede hacer un comentario más extenso encapsulado entre **/*** y ***/** . Con **BEGIN(comentario)** se indica donde se abre el comentario en la línea donde aparece **/*** y a continuación se definen las 3 posibles cosas que pueden aparecer, en primer lugar que aparezca un comentario seguido de ***/** , lo que indica el fin de comentario, en segundo lugar que aparezca una línea de comentario o nada (espacios en blanco) y por ultimo maneja el error de que no se haya cerrado debidamente el comentario, para ello se define la variable **inicio_comentario** que básicamente es un contador que indica en que línea se ha producido el error léxico.

También destacamos la definición de un identificador, que se construye a partir de la siguiente expresión regular: ***{carácter}({carácter}|{digito})**** en la cual vemos que un identificador puede construirse a partir de una letra y como añadido opcional n letras o dígitos concatenados a esa primera letra, el tamaño máximo de la cadena que define un identificador es de 16 caracteres, sobrepasar dicho tamaño sería considerado un error léxico y se maneja comprobando con *yyleng* el tamaño de la cadena. En el caso de que todo este correcto se guarda un duplicado de la cadena en *yylval.str*. Se define el token string con la siguiente expresión regular: **\"([^"\n]|\\.)*\"** y se almacena en *yylval.str* al igual que se hacía al definir el token ID. Por último, se manejan 2 posibles errores léxicos mediante expresiones regulares, se maneja la posibilidad de que se defina mal un string al entrecomillar mal una secuencia de caracteres con la siguiente expresión regular: ***\"([^"\n]|\\\")**** y haciendo uso de **pánico**, que como se explicó anteriormente detecta todo carácter no válido en el vocabulario del lenguaje, se maneja otro error léxico especificando que algo está incorrectamente escrito y en qué línea.

ANÁLISIS SINTÁCTICO

El analizador sintáctico se encarga principalmente de evitar la ambigüedad a la hora de analizar cadenas de símbolos de acuerdo con las reglas de una gramática formal. En el caso de este analizador sintáctico cabe remarcar el tratamiento de la ambigüedad, para ello se define en **sintactico.y** la asociatividad y la precedencia que tienen los operadores utilizando **%left PLUSOP MINUSOP**, **%left POR BARRAOP** y **%left UMINUS**, lo cual indica que se asocia por la izquierda en las operaciones de suma y resta y que tienen menor precedencia que la multiplicación y división que también se asocian por la izquierda, por último la negación, la cual tiene más precedencia que las anteriores operaciones.

Eso en cuanto a operadores y como se asocian, en cuanto a otras estructuras como el if-else tenemos otro tipo de ambigüedad, en este caso tenemos que resolver la asociatividad de cada if con su else, ya que al haber un if-else dentro de otro if-else surge el problema que hace que el compilador tenga que decidir de que if es cada else. Esto se traduce al problema que vimos en clase a la hora de realizar las tablas de análisis donde había que decidir en casos como este si desplazar o reducir, en nuestro caso se soluciona utilizando: **%expect 1**, lo que hace es suprimir la advertencia sobre conflictos que normalmente advierte Bison, pero como la mayoría de las gramáticas reales tienen conflictos inofensivos con desplazar/reducir que son fácilmente predecibles, a menos que cambie el numero de conflictos suprimo dichas advertencias.

En cuanto a la sintaxis que deben cumplir las expresiones se especifica al definir la propia **expression**. Por ejemplo: **expression : expression PLUSOP expression**. Otro ejemplo puede ser como declarar una variable o una constante, las cuales tienen la regla de producción **declarations : declarations VAR {tipo = VARIABLE;} identifier_list SEMICOLON | declarations CONST {tipo = CONSTANTE;} identifier_list SEMICOLON**. Luego esa declaración puede ser de 2 formas, o declarando el ID a secas o asignándole directamente a ese ID una expresión, además, se pueden declarar varias variables o constantes en una sola declaración separándolas por comas, eso lo reflejamos de la siguiente forma: **identifier_list : asig | identifier_list COMMA asig**, con esto conseguimos lo último mencionado, múltiples declaraciones en una sola línea separadas por comas y luego las otras 2 opciones que puede tener una asignación, asignar solo el ID o ID = expression: **asig : ID | ID ASSIGNOP expresión**.

En nuestra gramática también contamos con sentencias como print y read, que son definidas de la siguiente manera: **print_item: expression | STRING;** y **print_list: print_item | print_list COMMA print_item;** con las que podemos imprimir por pantalla una expresion/string o una lista de ellos; y **read_list: ID | read_list COMMA ID;** con la que podemos leer y asignar a una variable, o una lista de ellas, un valor pasado por consola.

Estas sentencias están definidas dentro de la sintaxis de los **statement**, en los que se encuentra también la definición de las sentencias if, if else, y while de la siguiente

manera: **IF LPAREN expression RPAREN statement ELSE statement | IF LPAREN expression RPAREN statement | WHILE LPAREN expression RPAREN statement** , donde LPAREN Y RPAREN son los token que hacen referencia a los parentesis. Como vemos en estos casos si se cumplen las condiciones descritas en una **expression** se pasan a ejecutar otros **statemen** o una lista de ellos.

ANÁLISIS SEMÁNTICO

El analizador semántico trabaja con una lista de símbolos con la que maneja los posibles errores que pueden darse, en primer lugar tenemos en la declaración de una función el manejo del posible error semántico que se produce si empieza de una forma distinta a esa regla, en segundo lugar tenemos en los identificadores que comprobar que no se redeclare ningún identificador haciendo una búsqueda en la lista de símbolos, si ese símbolo ya se encuentra en la lista se producirá un error semántico por redeclaración del identificador. En tercer lugar, tenemos lo mismo que ocurría para los identificadores pero en este caso para las constantes, una constante tampoco se puede redeclarar y por tanto se hace ese recorrido en la lista de símbolos para su comprobación y manejo del posible error semántico. En cuarto lugar, para las expresiones se maneja de cara a la redefinición de una variable que la expresión con la que se está trabajando sea de tipo variable y no sea una constante, ya que las constantes no se pueden modificar, manejamos ese posible error semántico comprobando el tipo del símbolo, también podría ocurrir que no se encontrase en la lista de símbolos la variable que buscamos, lo que ocasionaría otro error semántico. También a la hora de hacer un recorrido (for) debemos tener en cuenta lo mismo que para la redefinición de una variable, que el identificador que manejamos sea una variable y no una constante y que se encuentre en la lista de símbolos. En quinto lugar, a la hora de leer un identificador tenemos que hacer como ocurría al redefinir identificadores, hay que buscarlo en la lista de símbolos, comprobar que está en dicha lista y si no se produce un error semántico y si está en la lista comprobar que sea de tipo variable y no constante, si no se produciría otro error semántico. En sexto lugar, en el símbolo terminal **“id”** que hay en la gramática de **expression,** hay que comprobar que id esté en la lista de símbolos, si no, se producirá un error semántico y en el símbolo terminal **“id” “(“ arguments “)”** así mismo, se comprueba que id este en la lista de símbolos y además se comprueba al igual que en el primer caso que vimos al declarar una función que contemplaba el posible error semántico que se produce si empieza de una forma distinta a esa regla.

GENERACIÓN DE CÓDIGO

La generación de código se realiza utilizando la lista de código, en base a eso se traducen las operaciones a código máquina. Comenzamos con la traducción de los símbolos terminales de la gramática como son un **id** o un **num**, en el caso de cargar un identificador como símbolo terminal de **expression** se traduce como un **lw** donde la respuesta es un **obtenerReg()** y el primer argumento es un **concatena("_",$1)** que es el id (nombre de la variable) y como segundo argumento almacenamos NULL ya que no lo necesitamos. En el caso de **num** se traduce como un **li** donde el registro respuesta también se obtiene de **obtenerReg()** y el primer argumento es **$1** que es el propio número que estamos cargando en el registro. Al final de ambas operaciones se realiza un **insertaLC($$,finalLC($$),oper)**, para insertar dicha operación en la lista de código y se almacena el registro resultado de dicha lista de código con **guardaResLC($$,oper.res)**.

En el resto de las reglas gramaticales de **expression** que son la suma, resta, multiplicación, etc de una expresión con otra se traduce como un **add**, **sub**, **mul**, etc, en cada operación donde el registro resultado es **recuperaResLC($1)**, que es básicamente lo almacenado en la primera **expression**, como primer argumento lo mismo y como segundo argumento la otra **expression** a la que está sumando, restando o la operación que sea. Tras eso se inserta la operación en la lista de código como siempre, se libera la lista de código asociada al registro que actuaba de segundo argumento en la expresión aritmética y se libera el registro asociado a dicho argumento.

Los **read_list** se traducen como una serie de 3 operaciones, la primera es un **li** en el que se carga en el registro **$v0** un 5 para poder realizar el syscall, como segunda operación tenemos el propio syscall que acabo de mencionar y como tercera operación un **sw** donde cargan en **$v0** la **concatena("_",$1)**, que es básicamente _id. En el caso de la definición de un String también se traduce como una serie de 3 operaciones, en la primera se hace un **li** donde carga en **$v0** un 4 para el syscall, como segunda operación se hace un **la** donde se almacena **concatena2("$str",aux.valor)** en **$a0**, y por último se realiza el syscall mencionado anteriormente. En **print_item** se traduce de una forma similar, almacena 1 en **$v0**, hace un **move** donde almacena en **$a0** el argumento **recuperaResLC($1)**, que es básicamente el registro resultado que almacenaba la lista de código asociada a la expresión; finalmente realiza el syscall. Las operaciones que definen una variable o una constante se traducen como un **sw** donde como respuesta se almacena **recuperaResLC($3)** de la expresión que estamos almacenando; como primer argumento se almacena la **concatena("_",$1)** que es el id.

Un if se traduce en un par de operaciones, la primera operación que comprueba, en este caso un **beqz** donde como respuesta se almacena **recuperaResLC($2)** de la expresión y como primer argumento la etiqueta y tras esto como en todas las traducciones se inserta en la lista de código con: **insertaLC($$,finalLC($$),oper)**, se concatena la lista de código asociada a **expression** con la asociada a **statment** y se libera la lista de código asociada a este último; la segunda operación tiene como respuesta la etiqueta creada. Cuando a

este if-then se le añade un else, se hace exactamente lo mismo que en el anterior, pero hay una etiqueta más y 2 operaciones más, una que tiene como respuesta **etiq1** y la otra tiene como respuesta **etiq**. En el caso de los while se traducen también en 4 operaciones, la primera es la etiqueta que tiene como respuesta **etiq**, la segunda un **beqz** donde se comprueba la expresión y tiene como primer argumento **etiq1**, la tercera que tiene como respuesta **etiq** y la última que tiene como respuesta **etiq1**. Por último, remarcamos que en **program**, si todo está correcto se concatena **declarations** y **compound_statement** y se imprime el código de **declarations**, tras esto se libera su lista de código asociada.

FUNCIONES PRINCIPALES Y ESTRUCTURA DE DATOS

Al principio se comprueba si el archivo que queremos probar se puede abrir o no, después realiza una llamada al análisis sintáctico donde comprueba si el fichero contiene algún error, lo cual se comprueba en **lexico.l** y lo devuelve por pantalla; y también comprueba si hay errores semánticos y sintácticos a través de **sintactico.y** . Finalmente cierra el fichero.

El programa miniC en primer lugar detecta todos los elementos de código, donde se incluye las variables, las constantes con sus valores y los String con sus correspondientes valores. Cuando se detectan las variables, se comprueba que no hayan sido utilizado anteriormente y que sean de tipo variable como se explicó en el analizador semántico donde se manejaban dichos errores. Para las constantes se comprueba que no sean inmodificables.

A continuación, se incluyen la generación de código, donde dependiendo de si es una etiqueta devuelve el ope.res de su lista de código. Si no es una etiqueta, devuelve los valores de res, arg1 y arg2, si tienen valor.

Finalmente, introduce una operación li y un syscall para que a la hora de comprobar el código en spim, pueda tomarlo como bueno.

En el el fichero sintactico.y hemos definido una serie de métodos que utilizamos para realizar las comprobaciones y funcionalidades necesarias tanto en el análisis sintáctico como en el semántico.

Algunos de ellos son el método **ok()** que utilizamos para comprobar que no se ha producido ningún error de ningún tipo, en cuyo caso se continua y se pasa a imprimir el código ya en ensamblador.

También utilizamos métodos como **concatena()** para concatenar dos cadenas de caracteres o **obtenerReg()** y **obtenerEtiqueta()** para obtener los registros y etiquetas que no estén siendo utilizados.

Los métodos **perteneceTablaS**(), **nuevaEntrada**() y **esConstante**() sirven para realizar comprobaciones de la tabla de símbolos e insertar nuevas entradas en ella.

Por último, los métodos **imprimirTablas**() y **imprimirCodigo**() sirven para imprimir todas las declaraciones almacenadas en la tabla de simbolos y todo el codigo almacenado en la lista de código, ya transformado a lenguaje ensamblador.

Estructura de datos

Podemos encontrar una serie de operaciones que directamente se le da unos valores constantes como los string que obtienen valor 4 y siempre el mismo registro. Sin embargo, hay otros que hay que dárselos viendo que registro está libre, y cada vez que un registro ya se ha usado, liberarlo para no desperdiciar memoria. Otras para obtener su arg1, si tiene que hacer una concatenación de un carácter con un entero, que para eso se ha hecho una función encargada de ello.

Como hemos explicado anteriormente las declaraciones y el código se almacena en unas tablas que posteriormente se imprimen. Para ello utilizamos unas estructuras de datos llamadas ListaSimbolos y ListaCodigo.

En ListaSimbolos se define también un tipo de datos llamado **Simbolo** que será almacenado en esta lista y que está definido por un atributo cadena llamada “nombre”, un tipo y un entero llamado “valor”. El tipo es un enumerado que podrá ser variable, constante y cadena. El atributo valor solo se utilizará para almacenar el número de la etiqueta de cada cadena, que necesitaremos para imprimirlo más tarde, por lo tanto, es nulo cuando el símbolo se trata de una constante o una variable.

En ListaCodigo se defina el tipo de datos **Operacion** que contará con los siguientes atributos: un operador, que almacena que de qué clase de operación se trata, un registro, en el que se almacena por lo general el resultado de la operación, y dos argumentos, arg1 y arg2, que normalmente almacenan los registros en los que se encuentran los argumentos empleados en las distintas operaciones.

MANUAL DE USUARIO

En primer lugar, hay que hacer un make para generar los archivos correspondientes y compilar el propio compilador de miniC:

>> ***make***

Tras esto hay dos formas de ejecutar un programa con miniC, la forma más sencilla es que puede ejecutar directamente cualquier programa en miniC utilizando:

>> ***./mc.sh [nombreFichero]***

En segundo lugar, puede utilizar el makefile ejecutando:

>> ***make run***

Lo malo de utilizar la segunda opción es que ejecutara el fichero que esta puesto por defecto en el makefile, sin embargo, la primera opción tiene como ventaja que puede ejecutar el fichero que usted desee sin tener que modificar el makefile.

Si desea probar el compilador de miniC introduzca su propio archivo escrito en miniC o use los que hay por defecto en la carpeta pruebas.

EJEMPLOS DE FUNCIONAMIENTO

A continuación, mostraremos la salida correspondiente de la compilación de distintos ejemplos de código en miniC:

Fichero ***prueba_sintactico1.mc.txt***:

```
/************************
* Fichero de prueba nº 1
************************/
void prueba() {
// Declaraciones
const a=0, b=0;
var c=5+2-2;

// Sentencias
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";
}
```

Este fichero es el más completo ya que se usan la mayoría de las sentencias definidas en nuestra gramática. Además, no contiene ningún error léxico, sintáctico o semántico, por lo que deberá compilar sin problemas.

Al compilar este código en miniC se imprime la siguiente salida en código ensamblador:

```
##############################
.data
# STRINGS ###################
$str1: .asciiz "Inicio del programa\n"
$str2: .asciiz "a"
$str3: .asciiz "\n"
$str4: .asciiz "No a y b\n"
$str5: .asciiz "c="
$str6: .asciiz "\n"
$str7: .asciiz "Final"
$str8: .asciiz "\n"
# IDENTIFIERS ###############
_a: .word 0
_b: .word 0
_c: .word 0
####################
# Seccion de codigo
.text
.globl main
main:
li $t0,0
sw $t0,_a
li $t0,0
sw $t0,_b
li $t0,5
li $t1,2
add $t2,$t0,$t1
li $t0,2
sub $t1,$t2,$t0
sw $t1,_c
la $a0,$str1
li $v0,4
syscall
lw $t0,_a
beqz $t0,$l5
la $a0,$str2
li $v0,4
syscall
```

```
la $a0,$str3
li $v0,4
syscall
b $l6
$l5:
lw $t1,_b
beqz $t1,$l3
la $a0,$str4
li $v0,4
syscall
b $l4
$l3:
$l1:
lw $t2,_c
beqz $t2,$l2
la $a0,$str5
li $v0,4
syscall
lw $t3,_c
move $a0,$t3
li $v0,1
syscall
la $a0,$str6
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 $a0,$str7
li $v0,4
syscall
la $a0,$str8
li $v0,4
syscall
####################
# Fin
li $v0, 10
syscall
```

Fichero ***test_sem1.mc.txt***:

```
void test1() {
var a;
const b = 3, c = 0;
print "Test 1\n";
a = 1 + b;
read a;
print "Fin test1\n";
}
```

Salida sin errores de compilación:

```
##############################
.data
# STRINGS ###################
$str1: .asciiz "Test 1\n"
$str2: .asciiz "Fin test1\n"
# IDENTIFIERS ###############
_a: .word 0
_b: .word 0
_c: .word 0
####################
# Seccion de codigo
.text
.globl main
main:
li $t0,3
sw $t0,_b
li $t0,0
sw $t0,_c
la $a0,$str1
li $v0,4
syscall
li $t0,1
lw $t1,_b
add $t2,$t0,$t1
sw $t2,_a
li $v0,5
syscall
sw $v0,_a
la $a0,$str2
li $v0,4
syscall
###################
# Fin
li $v0, 10
syscall
```

Fichero ***test_sem2.mc.txt***:

```
void test2() {
var a;
const b = 3;
var a;
const a = 0;
a = 1 + b;
}
```

Este código en cambio sí que presenta errores, por lo tanto, al ejecutar el compilador, este informará de qué tipo de errores se tratan y no continuará con la compilación, por lo que no se imprimirá ningún código en ensamblador. La salida es la siguiente:

```
Linea 5: Variable a ya declarada
Linea 6: Variable a ya declarada
errores lexicos: 0, errores sintacticos: 0, errores semanticos: 2
Analisis semantico con errores
```

Fichero ***test_lex.txt***:

```
/*************************************
* Fichero de prueba de análisis léxico
*************************************/
void test2 () {
// Entero demasiado grande
const a = 123456789012;
// Identificador demasiado largo
_123456789012345678 = 1;
// Secuencia de caracteres incorrectos
print #@$%&;
// Cadena de caracteres sin cerrar
print "Hola\";
a = 1;
// Comentario sin cerrar
/* Esto no va a terminar nada bien
}
```

Este es otro ejemplo de un fichero con errores, aunque en este caso se tratan, además de errores léxicos, los cuales también serán notificados por el compilador. La salida es la siguiente:

```
ERROR: entero demasiado grande en linea 8: 123456789012
ERROR: identificador demasiado largo en linea 11: _123456789012345678
Linea 11: Variable _123456789012345678 no declarada
ERROR LEXICO: caracteres no validos en linea 14: #@$%&
Se ha producido un error sintactico en linea 14
Analisis sintactico con errores
Analisis semantico con errores
```