



Universitat
de les Illes Balears

COMPILADORES II

Práctica final

Alumnos:

Alejandro Mateo Fiol
Andrés Ramos Seguí
Jaime Crespí Valero

Profesor de la asignatura:

Dr. Pedro Antonio Palmer

Índice

Introducción	3
Código de tres direcciones	3
Códigos de operaciones	4
Tablas de variables, procedimientos y etiquetas	6
Tabla de variables	6
Tabla de procedimientos	6
Tabla de etiquetas	6
Código ensamblador	7
Optimizaciones	13
Optimización: Asignación diferida	14
Optimización: Cálculo de operaciones aritméticas	14
Optimización: Cálculo de operaciones lógicas	14
Optimización: Cálculo de condicionales	15
Casos de prueba	15
Caso de prueba 1	15
Caso de prueba 2	16
Caso de prueba 3	16
Caso de prueba 4	16
Caso de prueba 5	17
Caso de prueba 6	17

Introducción

La práctica consiste en el desarrollo de la parte back-end de un compilador que consiste en el código intermedio, optimización de código y código ensamblador. La práctica se realizará a partir de la parte front-end del compilador realizado en la primera parte de la asignatura. Por ello vamos a utilizar las mismas herramientas, es decir, utilizaremos la librería 'cup' para la gestión de las subrutinas semánticas que tendrán el código intermedio y de las clases de Java para la implementación del resto de funcionalidades.

El procesador tendrá que ser capaz de procesar el código fuente suministrado en un archivo de texto. Éste fichero se realizará a partir de la parte front-end del compilador.

A continuación se procederá a generar el código intermedio, optimizar el código si es posible y generar el código ensamblador resultante. También habrá de generar una serie de ficheros como resultado de su ejecución:

- Tabla de variables, procedimientos y etiquetas
- Fichero de código intermedio correspondiente al programa
- Fichero con el código ensamblador sin optimizar. Para cada instrucción de tres direcciones se mostrará un comentario con la instrucción y a continuación la traducción
- Fichero con el código ensamblador optimizado
- Errores, en el caso de que se detecten, se han de volcar en un fichero con los errores detectados, indicando el tipo de error, la línea y el tipo de mensaje explicativo

Las características principales que ha de tener:

- Se pueden hacer llamadas de subprogramas dentro de subprogramas
- Se pueden hacer declaraciones de subprogramas con parámetros

Código de tres direcciones

El código de tres direcciones es una notación que nos sirve para definir una notación de código que sea independiente del compilador. Esta notación será el paso intermedio entre el lenguaje fuente y el código ensamblador. Para ello, hemos definido una estructura denominada **Quadruple**, que se encuentra en la clase **Quadruple.java** que contiene:

- Código de la operación a realizar
- Operando 1
- Operando 2
- Operando destino

Como utilizamos el ensamblador 68K se ha definido ésta notación que va a ser la misma para todas las operaciones, que, en algunos casos no tendrá porqué tener un valor en todos sus operandos.

Códigos de operaciones

Los códigos asignados para definir las distintas operaciones son:

- **assign** → Equivalencia a asignar un valor o una variable a otra variable. Éste valor puede ser cualquiera de los tipos definidos.
- **procedureName** → Etiqueta que se coloca al principio de una función o procedimiento. Consiste en una etiqueta con el nombre del procedimiento
- **procedurePreamble** → Código de la inicialización que se ha de realizar al principio de cada función o procedimiento. El cual se encarga de generar el bloque de activación en la pila y de modificar el actual stack pointer y block pointer.
- **procedureReturn** → Código de la llamada return con el valor a retornar de una función.
- **procedureEnd** → Código que indica el final de una función o procedimiento.
- **procedureCall** → Código de la llamada **call**, que permite hacer una llamada a una función o procedimiento. Este se ocupa de escribir el valor de los parámetros en la pila.
- **procedureCallMain** → Código para llamar expresamente a la función main. La cual no tiene parámetros ni valor de retorno, es un caso único.
- **procedureParam** → Código para un parámetro de la función.
- **standardInput** → Nos permite introducir número por teclado
- **standardOutput** → Nos permite mostrar por pantalla valores o información.
- **sum, sub, mult, div** → Realiza las operaciones aritméticas más comunes: suma, resta, multiplicación, división.
- **and, or** → Realiza las operaciones lógicas AND y OR.
- **condTrue, condFalse** → Código que realiza un salto dependiendo de si se quiere comprobar que el valor es verdadero o falso.
- **equal, notEqual, greater, greaterOrEqual, lower, lowerOrEqual** → Realiza la evaluación lógica entre dos variables. El resultado debe ser un booleano.
- **skip, jump** → códigos de marcado de etiquetas y los saltos incondicionales para saltar a ellas.

Estructura de cada código de tres direcciones:

OPCODE	SOURCE_1	SOURCE_2	DESTINATION
assign	literal / id variable / id procedure	-	id variable
procedureName	-	-	id procedure
procedurePreamble	-	-	id procedure
procedureReturn	id procedure	-	id variable
procedureEnd	-	-	id procedure
procedureCall	-	-	id procedure
procedureCallMain	-	-	id procedure
procedureParam	id variable	número del parámetro	id procedure
standardInput, standardOutput	-	-	id variable
sum, sub, mult, div	id variable / literal	id variable / literal	id variable
and, or	id variable	-	id variable
condTrue, condFalse	id variable	-	id etiqueta
equal, notEqual, greater, greaterOrEqual, lower, lowerOrEqual	id variable	id variable	id variable
skip, jump	-	-	id etiqueta

Tablas de variables, procedimientos y etiquetas

En nuestro programa, dentro del paquete **BackendCompiler** y del fichero **TablesManager.java** podemos encontrar la definición de las tres tablas: tabla de variables, tabla de procedimientos y tabla de etiquetas.

Las tres tablas son un `ArrayList`, y almacenan en cada una información necesaria para realizar operaciones básicas (ocupación, desplazamiento,...). Todas estas tablas son impresas en un fichero `.txt` llamado **Tables_backend.txt**.

Tabla de variables

La tabla de variables es un `ArrayList` que contiene las variables que se va encontrando en el código de tres direcciones. Éstas variables que se almacenan se encuentran en el fichero **VariableBackend.java** y contienen:

- El nombre de la variable.
- El id del procedimiento del cual se ha llamado.
- El tamaño que ocupa.
- El offset en el que se encuentra dentro del bloque de activación.
- Su tipo subyacente básico (tipo de variable).

Tabla de procedimientos

La tabla de procedimientos es un `ArrayList` que contiene información de los procedimientos definidos en el programa. La información de estos procedimientos es almacenada en un objeto que se encuentra en la clase **ProcedureBackend.java** y contiene:

- El nombre del procedimiento.
- La profundidad en la que se encuentra.
- La etiqueta de inicio del procedimiento.
- El número de parámetros que necesita el procedimiento o función.
- El tamaño que ocupan sus variables locales.
- El tamaño que ocupan los parámetros introducidos.
- El tamaño que ocupa el procedimiento.
- El tipo subyacente básico que debe devolver la función. En caso de ser un procedimiento, éste atributo no se especificará.

Tabla de etiquetas

La tabla de procedimientos es un `ArrayList` que contiene la lista de etiquetas que han ido apareciendo en el código de tres direcciones. Éstas etiquetas han sido definidas en la clase **LabelBackend.java** y contiene únicamente el nombre de la etiqueta en cuestión.

Código ensamblador

A la hora de generar el código ensamblador utilizamos las macros creadas previamente a la creación del fichero `Assembler_code_not_optimized.X68` o `Assembler_code_optimized.X68`.

En cuanto a las funciones en ensamblador, todas tendrán el mismo nivel de profundidad y por lo tanto no habrá funciones definidas dentro de funciones. Hay que remarcar que tampoco tendremos variables globales, por lo tanto, deberemos utilizar variables locales.

A continuación se muestran las siguientes macros:

- **ASSIGNATION_INTEGER:** Asigna el valor de un integer (4 Bytes) en un espacio de memoria específico la pila de variables locales. Recibe los siguientes parámetros:
 - El valor del entero.
 - El offset dentro de las variables locales donde se ha de asignar el valor.
- **ASSIGNATION_BOOLEAN:** Asigna el valor de un booleano (2 Bytes) en un espacio de memoria específico de la pila de variables locales. Recibe los siguientes parámetros:
 - El valor del booleano.
 - El offset dentro de las variables locales donde se ha de asignar el valor.
- **ASSIGNATION_STRING:** Asigna el valor de una cadena de caracteres a una variable, en la cual cada carácter ocupa de (2Bytes). Recibe los siguientes parámetros:
 - El offset de la variable dónde se han de poner los caracteres del string.
 - Una variable que contiene la cadena de caracteres a copiar
 - El tamaño del string a copiar

Además, modifica los registros D0 y D1.

- **ASSIGNATION_VARIABLE_INTEGER:** Asigna el valor de una variable integer a otra variable integer. Recibe los siguientes parámetros:
 - El offset de la variable con el valor a asignar.
 - El offset de la variable destino
- **ASSIGNATION_VARIABLE_BOOLEAN:** Asigna el valor de una variable Boolean a otra variable Boolean. Recibe los siguientes parámetros:
 - El offset de la variable con el valor a asignar.
 - El offset de la variable destino.
- **ASSIGNATION_VARIABLE_STRING:** Asigna el valor de una variable . Recibe los siguientes parámetros:

- El offset de la variable dónde se han de poner los caracteres del string de destino.
- El tamaño restante que le queda al string de destino por llenarse.
- El offset de la variable que contiene el principio de la variable que contiene los caracteres del string de origen.
- El tamaño del string de origen.

Además, modifica el registro D0.

- **RETURN_GET_INTEGER:** Asigna el valor de return de una función a la variable especificada por el parámetro de entrada:
 - Offset de la variable donde se ha de asignar el valor del return.

Además, se modifican los registros A5, A6, A7.

- **RETURN_GET_BOOLEAN:** Asigna el valor de return de un procedure a la variable especificada por el parámetro de entrada.
 - Offset de la variable donde se ha de asignar el valor del return.

Además, modifica el registro D0.

- **RETURN_GET_STRING:** Asigna la cadena de caracteres de return de un procedure a la variable especificada por el parámetro de entrada.
 - El tamaño del string a retornar
 - Offset de la posición de inicio donde se ha de asignar el valor del return
 - El tamaño del string variable

Además, modifica el registro D0.

- **PUT_STRING_IN_PARAM:** Asigna la cadena de caracteres pasada por parámetro carácter a carácter en la pila de caracteres. Recibe los siguientes parámetros:
 - El offset del string de origen
 - El tamaño del string de origen
 - El tamaño restante del string de origen a asignar.

Además, modifica el registro D0.

- **STANDARD_INPUT:** Macro que recibe el offset de una variable donde almacenará un valor numérico leído por standard input. Llama al Trap #15 con la opción 4 (Read a number from the keyboard into D1.L) y lo asigna en el espacio de memoria especificado por el parámetro de entrada. Recibe el siguiente parámetro:
 - Offset de la posición donde se ha de almacenar el valor leído.

Además, modifica los registros D0 y D1.

- **OUTPUT_INTEGER:** Macro que muestra por pantalla (mediante la ayuda de la task 20 del trap #15), el valor de una variable entera especificada por parámetro (offset de la variable).
 - Offset de la variable con el valor entera
- **OUTPUT_BOOLEAN:** Macro que muestra por pantalla (mediante la ayuda de la task 20 del trap #15), el valor de una variable booleana especificada por parámetro (offset de la variable). Recibe el siguiente parámetro:
 - Offset de la variable con el valor booleano
- **PRINT_NEW_LINE:** Lee hasta que encuentra null. Macro que imprime el string hasta el final y pone un newline al final.
 - La dirección del buffer para imprimir por pantalla.
 - Offset de la variable a imprimir
 - El tamaño del string dividido en 2
- **RETURN_STRING:** Retorna el valor de una cadena de caracteres a una variable. Recibe los siguientes parámetros:
 - Offset del posición de memoria del return
 - Offset de la variable a asignar
 - Tamaño del string a asignar
- **ARITH_OPERATION_SUM:** Macro que realiza la suma de dos variables y la guarda el resultado en una tercera. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset del variable operando 1
 - Offset del variable operando 2
- **ARITH_OPERATION_SUB:** Macro que realiza la resta de dos variables y la guarda el resultado en una tercera. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset del variable operando 1
 - Offset del variable operando 2
- **ARITH_OPERATION_MULT:** Macro que realiza la multiplicación de dos variables y la guarda el resultado en una tercera. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset del variable operando 1
 - Offset del variable operando 2
- **ARITH_OPERATION_DIV:** Macro que realiza la división de dos variables y la guarda el resultado en una tercera. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset del variable operando 1
 - Offset del variable operando 2

- **LOGICAL_OPERATION_AND**: Macro que realiza una AND entre las dos variables recibidas por parámetro y guarda el resultado en el variable del primer operando. Recibe los siguientes parámetros:
 - Offset de la variable operando 1
 - Offset de la variable operando 2
- **LOGICAL_OPERATION_OR**: Macro que realiza una OR entre las dos variables recibidas por parámetro y guarda el resultado en el variable del primer operando. Recibe los siguientes parámetros:
 - Offset de la variable operando 1
 - Offset de la variable operando 2
- **COMPARISON_EQUAL_INT**: Macro que realiza una la comparación de dos variables enteras recibidas por parámetro. Si los dos valores son iguales, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

- **COMPARISON_EQUAL_BOOL**: Macro que realiza una la comparación de dos variables booleanas recibidas por parámetro. Si los dos valores son iguales, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

- **COMPARISON_NOT_EQUAL_INT**: Macro que realiza una la comparación de dos variables enteras recibidas por parámetro. Si los dos valores son diferentes, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

- **COMPARISON_NOT_EQUAL_BOOL**: Macro que realiza una la comparación de dos variables booleanas recibidas por parámetro. Si los dos valores son diferentes, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

- **COMPARISON_GREATER:** Macro que realiza una la comparación de dos variables enteras recibidas por parámetro. Si el valor del segundo parámetro es mayor que el valor del primero, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

- **COMPARISON_GREATER_OR_EQUAL:** Macro que realiza una la comparación de dos variables enteras recibidas por parámetro. Si el valor del segundo parámetro es mayor o igual que el valor del primero, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

- **COMPARISON_LOWER:** Macro que realiza una la comparación de dos variables enteras recibidas por parámetro. Si el valor del segundo parámetro es menor que el valor del primero, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

- **COMPARISON_LOWER_OR_EQUAL:** Macro que realiza una la comparación de dos variables enteras recibidas por parámetro. Si el valor del segundo parámetro es menor o igual que el valor del primero, guarda un 1, en caso contrario guarda un 0. Recibe los siguientes parámetros:
 - Offset de la variable de destino
 - Offset de la variable operando 1
 - Offset de la variable operando 2

Además, modifica los registros D0 y D1.

Para realizar el código ensamblador, realizamos un recorrido por toda la lista del código de 3 direcciones, traduciendo una a una cada una de las instrucciones en un string con el contenido en código ensamblador. Una vez tenemos el string con todo el código

ensamblador realizamos dos ficheros, uno para el código ensamblador sin optimizar (Assembler_code_not_optimized) y otro optimizado (Assembler_code_optimized).

En cuanto saltamos a una función, tenemos la etiqueta inicial al principio, la cual sigue el patrón de PROCEDURE_nombre_función. Seguido tenemos el preamble, el cual se ocupa de añadir nuestro bloque de activación en la pila, es decir, añadir los parámetros, el block pointer y el espacio para las variables locales. En este proceso, vamos a actualizar el block pointer al actual y poner el nuevo stack pointer.

Para llamar a un función, realizaremos diversas funciones. La primera de todas es reservar espacio para la variable de return, poner todos los parámetros en la pila para el siguiente bloque de activación, saltar a la función y por último recuperar el valor del return en caso de que precise.

Optimizaciones

La clase principal para las optimizaciones es 'CodeOptimizer', la cual recibe la lista de código de tres direcciones, seguidamente aplicará las optimizaciones y acabará devolviendo una nueva lista de código de tres direcciones ya optimizada.

A continuación haré una tabla con los métodos de la clase y su explicación:

Método	Parámetros	Devuelve	Explicación
setC3DList	-	-	Establece la lista de código de 3 direcciones que vamos a usar.
getC3DOptimized	-	Lista de código de 3 direcciones optimizada	Optimiza la lista de 3 direcciones y la devuelve. (Aplicará los métodos de abajo).
optimizeAssignments	Quadruple c3dInst int indexC3DInst	int	Optimización de asignaciones diferidas. Devuelve el índice de la lista de código 3 direcciones.
optimizeArithmeticOperations	Quadruple c3dInst int indexC3DInst	int	Optimización para operaciones aritméticas que son constantes. Devuelve el índice de la lista de código 3 direcciones.
optimizeLogicalOperations	Quadruple c3dInst int indexC3DInst	int	Optimización para operaciones lógicas que son constantes. Devuelve el índice de la lista de código 3 direcciones.
optimizeConditionals	Quadruple c3dInst int indexC3DInst	-	Optimización para el condicional If. Devuelve el índice de la lista de código 3 direcciones.

En cada una de las optimizaciones le pasaremos fundamentalmente dos parámetros. El primero es la propia instrucción de 3 direcciones que vamos a intentar optimizar. Para poder

hacer correctamente la optimización deberemos tener en cuenta las instrucciones del anteriores y posteriores a estar.

El segundo parámetro consiste en el índice de la misma instrucción en la lista de código de 3 direcciones y será usado en algunos casos para saber en qué posición de la lista nos encontramos. Por último, estos métodos devolverán el índice de la posición actual en la lista de código de tres direcciones. Esto es debido a que en las optimizaciones se eliminará instrucciones que son prescindibles.

Optimización: Asignación diferida

Esta optimización consiste en reducir al máximo las asignaciones más básicas. Para ello eliminaremos la variable temporal que almacena el valor de un literal y luego es asignada a la variable de verdad.

Código normal	Código de 3@ no optimizado	Código optimizado
x : int = 1.	temp = 1 x = temp	x = 1

Optimización: Cálculo de operaciones aritméticas

En esta optimización vamos a calcular las operaciones aritméticas, las cuales comprenden la suma, resta, multiplicación, división.

Código normal	Código de 3@ no optimizado	Código optimizado
x = (1 + 2 + 5).	temp1 = 1 temp2 = 2 temp3 = temp1 + temp2 temp4 = 5 temp5 = temp3 + temp4 x = temp5	x = 8

Optimización: Cálculo de operaciones lógicas

En esta optimización vamos a calcular las operaciones lógicas, las cuales comprenden la en and (&&) y el or (||).

Código normal	Código de 3@ no optimizado	Código optimizado
x = (true false).	temp1 = true temp2 = false temp3 = true false x = temp3	x = false

Optimización: Cálculo de condicionales

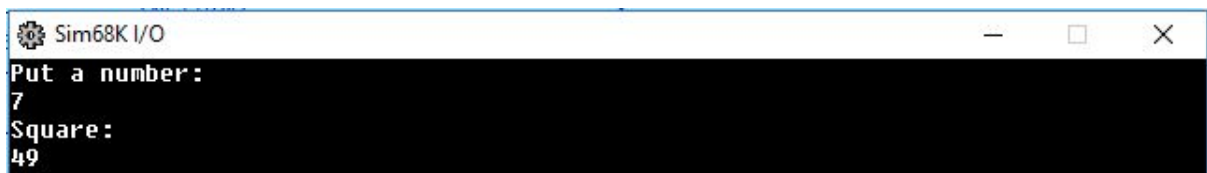
En esta optimización vamos a optimizar el condicional del 'IF'. El condicional que tenemos por defecto es poco eficiente a la hora de acabar. Cuando es verdadera una de las condiciones de nuestro condicional, entonces entrará y hará las sentencias correspondiente para luego acabar en una instrucción de 'goto' que hará un salto. Este salto no irá directamente al final como debería ser la versión más mejorada, sino que irá haciendo saltos hasta llegar al final. Por ello hemos optimizado el condicional para que al acabar la sentencia pueda saltar directamente al final del condicional.

Casos de prueba

Hemos creado 6 casos de prueba para comprobar el correcto funcionamiento de la práctica. Los casos de pruebas los vamos a separar en dos grupos. El grupo de pruebas para comprobar el correcto funcionamiento del código y los casos de prueba para comprobar la detección de errores en tiempo de compilación por parte de nuestro compilador. Todos los casos de prueba se podrán encontrar en la carpeta de 'output' en el código fuente.

Caso de prueba 1

En este caso vamos a testear el correcto funcionamiento de las funciones con y sin parámetros, también vamos a comprobar como las funciones devuelven el valor al realizar el return. En esta prueba el correcto funcionamiento de la entrada por teclado (input) y de la salida por teclado (output). Además de lo anterior, vamos a utilizar los tipos primitivos (int, boolean y string) y vamos a hacer cálculos aritméticos con los enteros.



```

Sim68K I/O
Put a number:
7
Square:
49

```

Caso de prueba 2

En este caso de prueba vamos a comprobar el correcto funcionamiento del bucle while, de los operadores relacionales y el uso de múltiples operaciones aritméticas y múltiples parámetros en funciones.



```
Sim68K I/O
Result factorial:
1920
```

Caso de prueba 3

En este caso de prueba vamos a comprobar el funcionamiento del condicional If y de los operadores lógicos. Este caso de prueba ha sido preparado para poder testear las diferentes optimizaciones.



```
Sim68K I/O
Num1 is bigger
8
```

Caso de prueba 4

Empezamos con los casos para comprobar los errores. La asignación de un string más grande a uno más pequeño conllevará un error ('AssignmentSizeOverflowException') ya que los strings son estáticos y por tanto, tiene un tamaño fijo. El fichero para esta prueba es el 'fallo1.txt'.

```
Exceptions.AssignmentSizeOverflowException: Exception: Compilation error.
Message: Error al intentar asignar un string de mayor tamaño a uno de menor tamaño.
La variable fuente es #string2, con tamaño 30 (en bytes).
La variable destino es #string1, con tamaño 6 (en bytes)..
|   at BackendCompiler.AssemblerConverter.getAssignment(AssemblerConverter.java:349)
|   at BackendCompiler.AssemblerConverter.generateAssemblerCode(AssemblerConverter.java:87)
|   at BackendCompiler.BackendManager.generateAssemblerCode(BackendManager.java:51)
|   at BackendCompiler.BackendManager.generateAssemblerCodeWithoutOptimization(BackendManager.java:55)
|   at practiacompiladores.Parser$CUP$Parser$actions.CUP$Parser$do_action_part000000000(Parser.java:588)
|   at practiacompiladores.Parser$CUP$Parser$actions.CUP$Parser$do_action(Parser.java:3661)
|   at practiacompiladores.Parser.do_action(Parser.java:398)
|   at java_cup.runtime.lr_parser.parse(lr_parser.java:699)
|   at practiacompiladores.Main.executeAnalyzer(Main.java:134)
|   at practiacompiladores.Main.main(Main.java:58)
```

Además de darnos el error, se generará un fichero llamado 'Compilation_error.txt'.

Caso de prueba 5

En este caso de prueba vamos a comprobar como nos da la excepción 'StringSizeOverflowException' cuando un string supera el tamaño máximo de 255 caracteres, el cual es el límite para el ensamblador de 68k. El fichero para esta prueba es el 'fallo2.txt'.

```
Exceptions.StringSizeOverflowException: Exception: Compilation error.  
3) Exit.  
Name: StringSizeOverflowException  
Message: Your string is too large. It has 256 characters and must have lower than 255.  
    at practiacompiladores.Parser$CUP$Parser$actions.CUP$Parser$do_action_part000000000(Parser.java:1702)  
    at practiacompiladores.Parser$CUP$Parser$actions.CUP$Parser$do_action(Parser.java:3661)  
    at practiacompiladores.Parser.do_action(Parser.java:398)  
    at java_cup.runtime.lr_parser.parse(lr_parser.java:699)  
    at practiacompiladores.Main.executeAnalizer(Main.java:134)  
    at practiacompiladores.Main.main(Main.java:58)
```

Caso de prueba 6

En este caso vamos a detectar en tiempo de compilación una división con denominador cero. Para ello lanzaremos la excepción 'DivisionByZeroException', la cual se muestra por pantalla y se almacena en un fichero llamado 'Compilation_error.txt'. El fichero para esta prueba es el 'fallo3.txt'.

```
Exceptions.DivisionByZeroException: Exception: Compilation error.  
Name: DivisionByZeroException  
Message: You can't divide by zero. Your operation 1/0 is invalid.  
    at practiacompiladores.Parser$CUP$Parser$actions.CUP$Parser$do_action_part000000000(Parser.java:1350)  
    at practiacompiladores.Parser$CUP$Parser$actions.CUP$Parser$do_action(Parser.java:3661)  
    at practiacompiladores.Parser.do_action(Parser.java:398)  
    at java_cup.runtime.lr_parser.parse(lr_parser.java:699)  
    at practiacompiladores.Main.executeAnalizer(Main.java:134)  
    at practiacompiladores.Main.main(Main.java:58)
```