Autores: Diego Calderón y Erwin Morataya

Guatemala 21 de Noviembre de 2014

COMPILADOR CC4

# Documentacion Fase 0 / Command Line Interface - CLI

El archivo Compiler.java tiene el main y recibe los parametros de compilacion para la ejecución y esto se recibe como un arreglo en el main, utilizamos unas variables de configuracion del input, output, target, optimizaciones y degub las cuales se les asigna la opcion seleccionada para cada bandera, luego se validan los valores y que el archivo de input tenga extencion .dcf dado que asumimos que como es un compilador del lenguaje Decaf esta sera la extencion del archivo de input necesaria, la ayuda se implemento como un metodo de la clase Compiler.java y se creo un metodo de exit para que cierre el programa para simplificar el codigo.

Despues de haber configurado las variables y validar los datos se creo una instancia de cada Fase hasta al target seleccionado si no se indica que el target sea codegen entonces no se realizara la fase de optimizacion dado que esta es la ultima fase. Todos los Constructores reciben un objeto de la fase anterior excepto el de Scanner que recibe el nombre del archivo de input, se verifica que fases se deben debuggear y se les pasa un objeto Debug que se encarga de imprimir los mensajes de Debug.

Los errores se manejan como excepciones por lo que la clase ErrorHandler.java hereda de Exception y cuando ocurre se lanza una excepcion con un mensaje personalizado y esta se propaga hasta el main y se despliega solo en el main.

Para imprimir al archivo se agrego una clase OutputFile.java en el paquete lib la cual maneja la salida hacia un archivo y esta se le pasa como parametro a todas las fases para que puedan guardar en el archivo de output.

# Documentación fase 1 / Scanner

Se creó un Lexer temporal adicional para poder recorrerlo y dejar el original para que lo reciba el parser y no se produzca un error de que el lexer no tiene mas tokens y ya llegó al `EOF`.

El lexer es recorrido en un ciclo `while` hasta llegar al `EOF` en el ciclo se obtiene el siguiente token y se obtiene el tipo que retorna un numero entero y en un arreglo llamado `ruleNames` en el archivo `DecafLexer.java` y se despliega a pantalla si el Debug esta activo y se guarda en el archivo de salida.

Si el token es una de las reglas de error entonces de despliega un mensaje con el error.

Se removieron los ErrorListener que tiene antlr por defecto y se agrego uno personalizado se dejo como un campo de la clase `Scanner.java` para que el Parser tambien pudiera usarlo

### Errores

En esta parte de la fase se contemplaron errores en `Strings`, `Chars`, `hexadecimales` y un caso de error general

* En los `strings` se contempló que los caracteres de escape sean validos y que el string finalice con una comilla doble.
* En los `char` se contempla que se pueden tener caracteres de escape que no sean validos y que se debe finalizar con una comilla simple.
* En el caso de `hexadecimal` se contemplo que `0x` sería un error o si tiene letras fuera del rango de la a-fA-F.

# Documentación fase 1 / Parser

El parser recibe como parámetro en el constructor un objeto tipo `Scanner` y obtiene el campo `lexer`, para crear el DecafParser se removieron los ErrorListeners por defecto y se usó el mismo BaseErrorListener de Scanner.

En cada regla de derivación se despliega en pantalla que regla es.

Se crearon reglas de derivación para los errores y cada regla de error tiene asociada un `notifyErrorListeners(msg)` donde `msg` es un String con la descripción del error.

La gramática fue escrita en base al documento `decaf-specifications.pdf` y se fue modificando para desplegar los errores.

Se modificaron los testcase dado que los testcase contenían `class` pero las especificaciones de decaf no indicaban que el código debía estar contenido en clases.

### Errores

En esta parte de la fase se contemplaron más errores según los testcase provistos, los mensajes de error son desplegados con la descripción del error y el número de línea donde se encuentra.

**Declaración de variables:** Se verificó que solo se puede tener dos tipos `boolean` o `int` que pueden ser un arreglo perosolo se puede inicializar el número de elementos del arreglo con un numero entero, no se puede asignar un valor en la declaración, se puede declarar varias variables separadas por coma siempre y cuando sean del mismo tipo, se verifico que tuvieran la separación por coma, se verificó que se haya escrito el identificador de la variable y no sea una palabra reservada.

**Ciclos for:** Se verificaron los paréntesis y los parámetros que debe llevar dentro de los paréntesis, además también las llaves de apertura y cierre del bloque de código que pertenece al `for.

**Condición if:** Se verifico que tenga una condición y que tenga los paréntesis de apertura y de cierre y las llaves de apertura y cierre del bloque.

**Ciclo while:** Se verificó que contenga la condición del ciclo y los paréntesis.

**Estructura del programa:** Se verificó el orden de la estructura del programa según las especificaciones de Decaf.

#### Actualización

**Scanner**: Se corrigió la gramática para los comentarios.

**Parser:** Se cambiaron el orden de algunas reglas por problemas en la jerarquía de operaciones matematicas.

# Documentación fase 2 / AST

- Se utilizaron las siguientes etiquetas para generar los visits con `ANTLR` y se agregó la bandera `visitor` para que se generara el archivo `DecafParserBaseVisitor.java` del cual se heredó para sobrescribir los métodos de visita de las reglas y en este crear los nodos con la información que nos interesa guardar para las siguientes fases.

* `# root`
* `# asignacion`
* `# ifs`
* `# ciclo`
* `# statement`
* `# exp`
* `# literales`
* `# operacion`
* `# par`
* `# bloque`
* `# method\_dec`
* `# method\_c`
* `# field\_dec`
* `# error`

- Para poder realizar esta fase fue necesario retirar todo el código de java que se encontraba en los archivos de gramáticas `DecafLexer.g` y `DecafParser.g` y se intentó trasladar la notificación de errores del parser al `DecafParseBaseListener.java` pero no funcionó como se deseaba y para no retrasarnos se postpuso la notificación de errores y se avanzó utilizando casos ideales sin errores, posteriormente se completará la notificación de errores.

- El árbol de parseo generado por `ANTLR` se despliega utilizando el `TreeViewer` el cual se agrega a un `JFrame` de la librería swing para poder desplegarlo gráficamente en una ventana.

- En esta fase siempre se despliega el árbol en la terminal pero no nos dio tiempo de implementar el debug y que solo cuando estuviera activo.

## Nodos

Los siguientes nodos se crearon heredando de la clase abstracta `Node` y nos van a servir para guardar la información que nos interesa y formar el AST por medio de una lista de estos nodos que van a estar en la clase `Root`. La clase `Root` también nos va a servir para implementar los bloques.

* 'Root'
* 'Literal'
* 'Cond'
* 'BinOp'
* 'Cycle'
* 'Exp'
* 'MethodCall'
* 'VarLiteral'
* 'Statement'
* 'Asign'

# Documentación fase 3 / Semantic

- Para crear nuestra tabla de símbolos implementamos una clase `Table` que contiene el nombre de la tabla, el padre de la tabla y una HashTable que contiene el nombre de la variable o método, el tipo y si es método contiene también el tipo de cada uno de sus parámetros. También implementamos la clase `SymbolTable` que va a contener todas las tablas creadas en forma de una lista y se creo un método para poder buscar cada tabla por medio de su nombre.

- Para hacer las verificaciones de unicidad, existencia y tipos realizamos un método `check` para cada clase que se encarga de hacer estas verificaciones, en la clase semantic en donde recibimos un ast obtenemos el root que contiene los nodos del ast y verificamos cada nodo para ver a que clase pertenece y aplicarle su método de verificación. Para cada uno de estos métodos de verificacién se pasa como parámetro la tabla, la lista de todas las tablas y una lista de errores y para verificar la unicidad y existencia de las variables declaradas se revisa cada tabla de la lista por medio del padre de la tabla. Para la verificación de tipos se realizaron métodos para las expresiones que nos van a dar como resultado un string y ese string va ser el tipo de la expresión y ese string se compara con el tipo de la variable a la que se le esta asignando la expresión para ver si corresponden, este string además puede ser de tipo "error" que nos indica que se estan operando ints con booleans o que los tipos no corresponden con la operación. Este string también nos sirve para la verificación de la condición de un if y while, la inicialización del for, el tipo de return de un método y verificaciones de este tipo.

- La lista de errores se le pasa a los métodos `check` en donde se pueden generar errores para poder agregarlos al listado y posteriormente desplegarlos.

- Se verifico que los arreglos no puedan ser de 0 posiciones, no se verifico que fueran positivos porque por la gramatica solo se pueden tener números mayores o iguales a 0.

## Ejemplo

int a;

int add(int a, int b) {

return a + b;

}

int main() {

a = add(2, 3);

return a;

}

Este código generaría las siguientes tablas de símbolos:

**Parent:** null

**Name:** ROOT

|  |  |  |
| --- | --- | --- |
| Nombre | Tipo | Tipo de parametros |
| a | Int | Null |
| add | Int | {int, int} |
| Main | int | Null |

**Parent:** ROOT

**Name:** add

|  |  |  |
| --- | --- | --- |
| Nombre | Tipo | Tipo de parámetros |
| a | Int | Null |
| b | int | Null |

- no hay declaraciones de variables en main por lo tanto no tiene tabla de símbolos

## Errores:

Este función debería ser valida pero tenemos un error en la verificación del return que reporta ese programa como invalido nos dimos cuenta al finalizar Irt/codegen y no teniamos suficiente tiempo para solucionarlo

int factorial(int n) {

if (n <= 1) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

# Documentacion fase 3 / Irt-Codegen

- Se realizaron las dos fases juntas para agilizar el tiempo de desarrollo dado que solo generamos código para `MIPS`.

- Se modifico parte de lo desarrollado para la fase de semantic.

- Se implementaron metodos destruct para todos los nodos del AST, por comodidad y orden se movieron las clases que heredan de `Node` a un subpaquete llamado `nodes`.

Para el manejo de registros se crearon las clases

* `Register`: Contiene un campo con el numero de registro y el tipo (s o t)
* `RegisterManager`: Administra el uso de los registros posee un metodo para otorgar registros y otro para recibir registros que se han dejado de usar para evitar el uso de un registro dos veces y se tenga perdida de datos.

Se creó una clase abstracta `Instruction` de la que heredan todos los tipos de instrucciones:

* `Alu`: esta instruccion abarca todos los tipos de instruccion en la que se realicen operaciones matematico logicas(suma, resta, multiplicacion, division).
* `Immediate`: Pendiente de uso
* `Jump`: saltos a etiquetas o registros (jal, jr, j)
* `Label`: indica un label en el codigo
* `LoadStore`: sw, lw, sb, lb.
* `Syscall`: llamada a funcion del sistema de MIPS

Se implementó una funcion `print(int n)` para poder desplegar resultados, se agregó a la tabla de simbolos en semantic para que no fuera una función invalida para las verificaciones de semantic.

print:

add $sp $sp -8

sw $a0 0($sp)

sw $v0 4($sp)

li $a0 0($fp)

li $v0 1

syscall

lw $a0 0($sp)

lw $v0 4($sp)

add $sp $sp 8

jr $ra

### Errores:

* Cuando se accede a una posición por medio de una variable da error como en el siguiente ejemplo:

int array[10], n;

n = 5;

print(array[n]); // aqui se produciria un error