Universidad del Valle de Guatemala. Semestre 1, 2018. Departamento en Ingeniería en Ciencias de la Computación y Tecnologías de la Información.

# Fase No. 3, Proyecto – Construcción de Compiladores

BROLO, G.
Carnet 15105

Documentación presentada para el curso de CONSTRUCCIÓN DE COMPILADORES, S. 10, Universidad del Valle de Guatemala.

---

**Abstract**

Este proyecto abarca la implementación de un sistema de tipos para el lenguaje decaf, así como la implementación de una tabla de símbolos y la implementación de las reglas semánticas para el mismo lenguaje[1], en pro de la construcción de un compilador para el lenguaje Decaf. También, se incluye la generación de código intermedio a través del uso de código de tres direcciones. Asimismo, se genera código de bajo nivel, en lenguaje MIPS como última fase del compilador, incluyendo algunas optimizaciones. Por último, se integran estas funcionalidades a través de un IDE. Se utilizó el lenguaje Java como lenguaje principal para la construcción del compilador, el framework Vaadin para la interfaz gráfica del IDE y ANTLR 4 para la construcción del árbol sintáctico y el recorrido de este para la implementación del sistema de tipos y las reglas semánticas. Se generó lenguaje MIPS de bajo nivel como el lenguaje objetivo de este compilador.

**Objetivos**

Implementar el conjunto de reglas semánticas descritas en el proyecto, así como implementar el sistema de tipos propuesto. Agregar las acciones semánticas al analizador sintáctico y construir la tabla de símbolos del compilador. Generar código de tres direcciones para el código de alto nivel ingresado por el usuario. Generar código MIPS que provenga del código de tres direcciones generado. Realizar un IDE gráfico para la implementación de pruebas.

[1] Obtenidas de la guía de la fase 1 del proyecto.

**Analizador sintáctico**

El conjunto de reglas semánticas aplicadas es: ningún identificador se puede declarar dos o más veces en un ámbito, ni puede ser utilizado sin ser antes declarado. Existe un método *main* en cada programa, de lo contrario el programa no puede compilarse. Al declarar arreglos, *num* (el valor del tamaño del arreglo) es un entero mayor a cero. El número y tipo de parámetros en la llamada a un método debe coincidir con la definición del método, i.e. cada parámetro corresponde al tipo y se debe proporcionar la cantidad de parámetros que es.

Si el método es de tipo *void*, no existe *return*, de lo contrario, debe existir un *return* con el mismo tipo que debería devolver el método. Dentro de un *if* y *while*, la condición es de tipo *boolean*. Los operadores aritméticos y relacionales solo pueden operarse con operandos de tipo *int*, mientras que los operandos para los operadores de igualdad deben ser *int*, *char* o *boolean* y los operandos deben de ser del mismo tipo. Los operandos para los operadores condicionales son de tipo *boolean*. El valor del lado derecho en una asignación debe corresponder con el tipo de la *location* del lado izquierdo.

Dentro del sistema de tipos se encuentran las siguientes operaciones: existe un tipo de dato llamado *int*, existe otro tipo de dato llamado *char*, existe otro tipo de dato llamado *struct*, existe un tipo de dato llamado *boolean* y finalmente, existe un tipo de dato llamado *void*. Dados A y B, con tipo *int*, A y B pueden ser operados con los operadores +, -, /, * y %, devolviendo un dato C de tipo *int*.

Dados A y B, con tipo *int*, A y B pueden operarse con los operadores >, <, ==, !, >= y <=, devolviendo un dato C de tipo *boolean*. Dados M y N con cualquiera de los tipos existentes, M = N es una operación válida sí y solo si M y N son del mismo tipo; la operación devuelve un tipo de dato *void*. Existe un dato *true*, y es de tipo *boolean*. Existe un dato *false*, y es de tipo *boolean*. Para la estructura del *if*, el tipo del parámetro A es de tipo *boolean*. Para la estructura while, el tipo de parámetro es boolean. A continuación, se muestra la definición del sistema de tipos empleado:

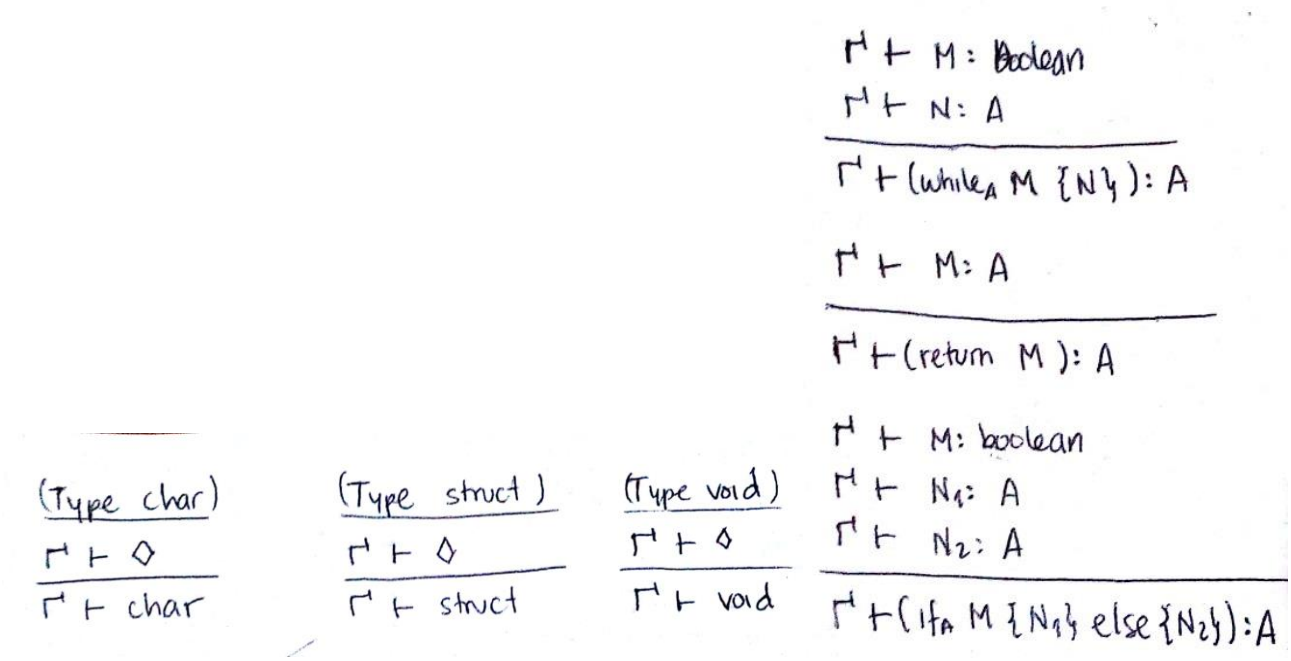


*Figura 1. Parte del sistema de tipos empleado.*

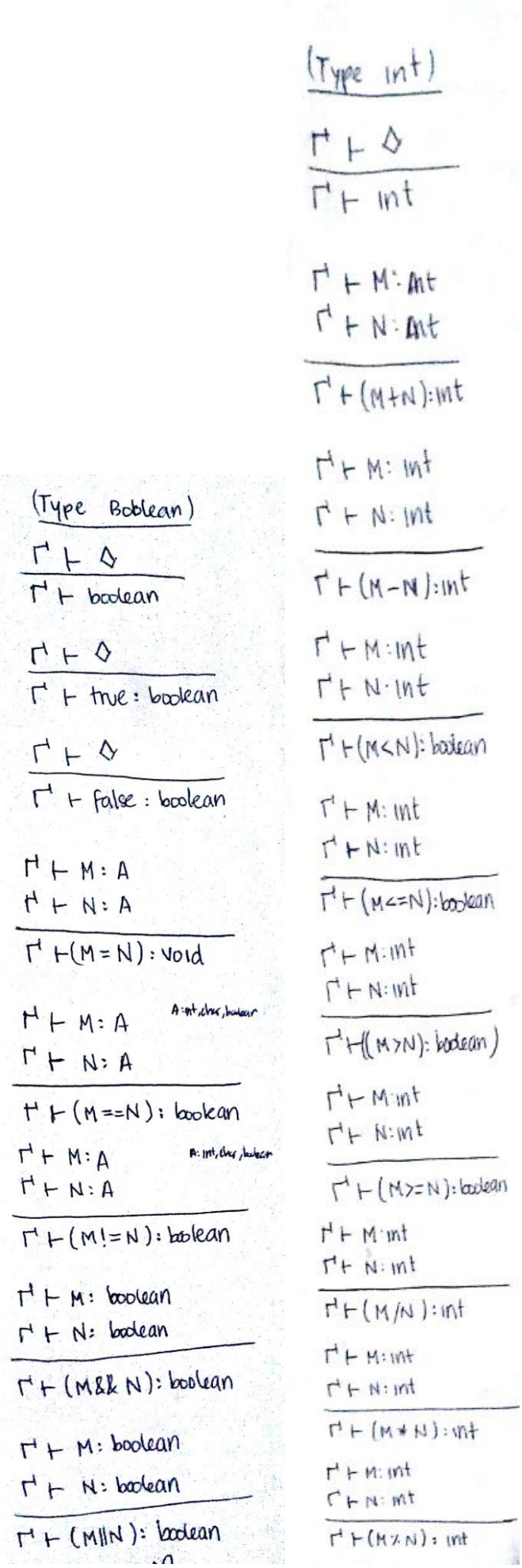


*Figura 2. Otra parte del sistema de tipos empleado.*

Se utilizó el lenguaje Java 8 para la construcción del proyecto. El *framework* web *Vaadin* para la interfaz gráfica y la presentación de resultados de compilación. Se utilizó ANTLR 4 para la generación del árbol sintáctico. Se utilizó un *listener* de ANTLR para recorrer el árbol e ir verificando las acciones semánticas de la gramática y para implementar el sistema de tipos. La tabla de símbolos se implementó como una lista que guarda objetos de diversos tipos; existen objetos de tipo Variable y tipo Método, así la tabla sabrá que tipo esta guardando y luego se podrá verificar que las implementaciones del sistema de tipos y las reglas se den con los tipos correctos. Para el tipo Struct se hizo una extensión del tipo de objeto de Método, verificando además cada objeto de tipo variable dentro de su declaración.

El flujo común del programa se encuentra plasmado en el siguiente diagrama:

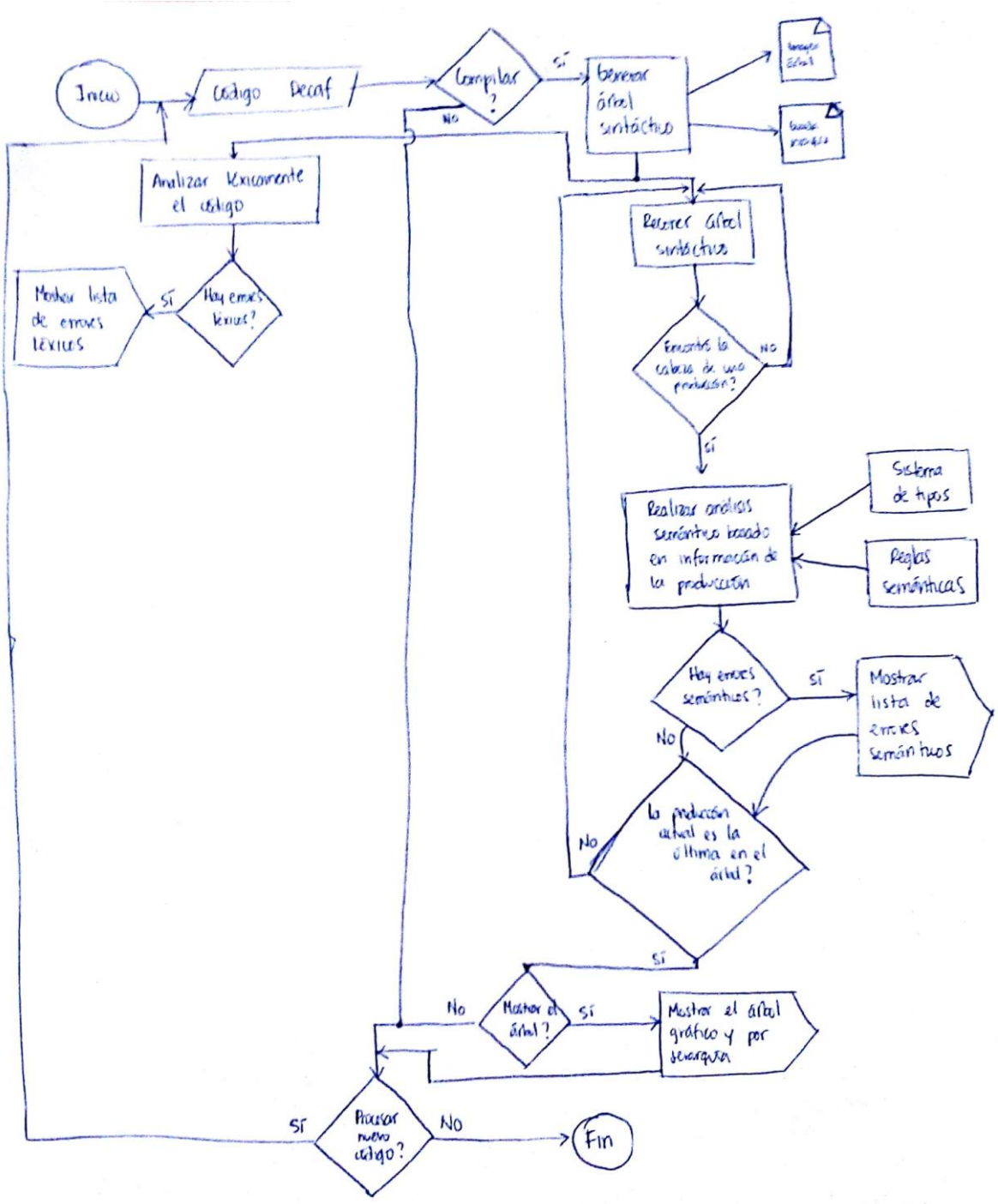


*Figura 3. Flujo del programa..*

**Generación de código intermedio**

Se utilizó el código de tres direcciones (TAC) como el lenguaje intermedio para este compilador. Este se basa en instrucciones con, a lo sumo, tres operandos. Cuando una expresión tiene más de tres operandos, se agregan variables temporales y se divide nuevamente la instrucción en instrucciones de tres operandos. Para la **asignación de variables** se permite:

- var = constant;
- var1 = var2;
- var1 = var2 op var3;
- var1 = constant op var2;
- var1 = var2 op constant;
- var = constant1 op constant2;

Los operadores permitidos son +, -, *, / y %. Los operadores booleanos permitidos son <, ==, | | y &&. Para el flujo de control if, la estructura del código es la siguiente:

```
_t0 = condition;
IfZ _t0 Goto _L0;
successfull if;
Goto _L1;
 _L0:
else part;
_L1: z = z * z;
```

El flujo de control while tiene la siguiente estructura:

```
L0:
 _t0 = condition;
IfZ _t0 Goto _L1;
code for unmet condition;
Goto _L0;
_L1:
code when condition is met;
```

**Cambios en el diseño**

Para los métodos, se debe utilizar la cláusula BeginFunc N, en dónde N es el espacio que se reservará para las variables locales y temporales, calculada en bytes; para finalizar un método, se debe utilizar la cláusula EndFunc. Cuando se tiene una llamada a un método se debe utilizar la cláusula PushParam paramlist, en dónde paramlist es la lista de parámetros a ingresar, separados por coma. La llamada se realiza con la cláusula LCall methodName; posteriormente, se debe escribir PopParams N, donde N es el espacio que fue utilizado en variables locales y temporales y servirá para regresar el stackframe a su posición original antes de la llamada al método.

Para implementar la generación de código, se adicionaron nuevos métodos al diseño anterior. Se creó un método *getNextTemp( )* que devuelve la siguiente variable temporal a utilizar en un determinado scope. La función *assignTemporals( )* asigna una variable temporal a una expresión calculada. Este cálculo es realizado al mismo tiempo

que se ejecuta el análisis sintáctico. Luego, una función writeAssignTAC toma la lista de expresiones TAC generada y la escribe en el archivo *decaf.tac*.

**Pruebas del analizador sintáctico**

Se realizaron diversas pruebas para verificar la propia implementación de las reglas semánticas y el sistema de tipos. Los archivos de prueba se encuentran en el directorio del proyecto bajo la ruta: *examples/*.

**Generación de código objeto: MIPS**

Se eligió el lenguaje MIPS de bajo nivel como el lenguaje objeto a generar en este compilador. Se eligió este mismo debido a su amplio set de instrucciones y facilidad de integración desde el código de tres direcciones. Se creó un nuevo parser para analizar el código de tres direcciones generado, por lo que se creó un nuevo Listener, *TacSemanticListener*, para recorrer los tokens generados y poder escribir nuevo código en MIPS. Se crearon tres métodos para obtener los registros disponibles para el código a generar: *getNextAvailableTR, getNextAvailableSVR, SVR,* y *getNextAvailableAR*. El primero para obtener registros temporales, que en mips van desde *$t0* hasta *$t7*, el segundo para obtener registros de valores guardados (registros *$s*) y el último para obtener registros de argumentos.

La asignación de los registros se basa en el código intermedio y se traducen directamente estos enunciados a código en MIPS. Para las variables que no son registros temporales o algún otro registro de MIPS, se les trata como *.word*, variables que están guardadas en memoria, bajo la sección de *.data* de MIPS. El archivo generado se compone de dos secciones, la sección *.text*, que contiene todas las funciones y la función main como la función principal, y la sección de datos, con los datos de las variables comentadas anteriormente.

**Resultados y comentarios**

Se implementaron las reglas semánticas de la guía del proyecto y el sistema de tipos propuesto. Se construyó además un IDE web para poder compilar código en Decaf. Por esto mismo, se cumplió con las funcionalidades propuestas en la guía del proyecto. Se podría mejorar en la presentación de algunos errores, ya que hay errores de css en dónde algunos errores no se ven por completo en el área de la consola.

Durante la presentación se encontró que varias reglas no se habían terminado de implementar. Esto se debió en su mayor parte a que no se habían entendido por completo las reglas y se pensó que se habían terminado de implementar, en algunos casos y en otros casos no se consideraron casos pequeños en donde estas reglas podían fallar. Sin embargo, los errores ya fueron arreglados. En un futuro sería mejor estar totalmente seguros de haber entendido qué funcionalidad es la que se implementará.

Con fines de mostrar brevemente el IDE y algunas ejecuciones, se muestra a continuación, un ejemplo de una ejecución exitosa:

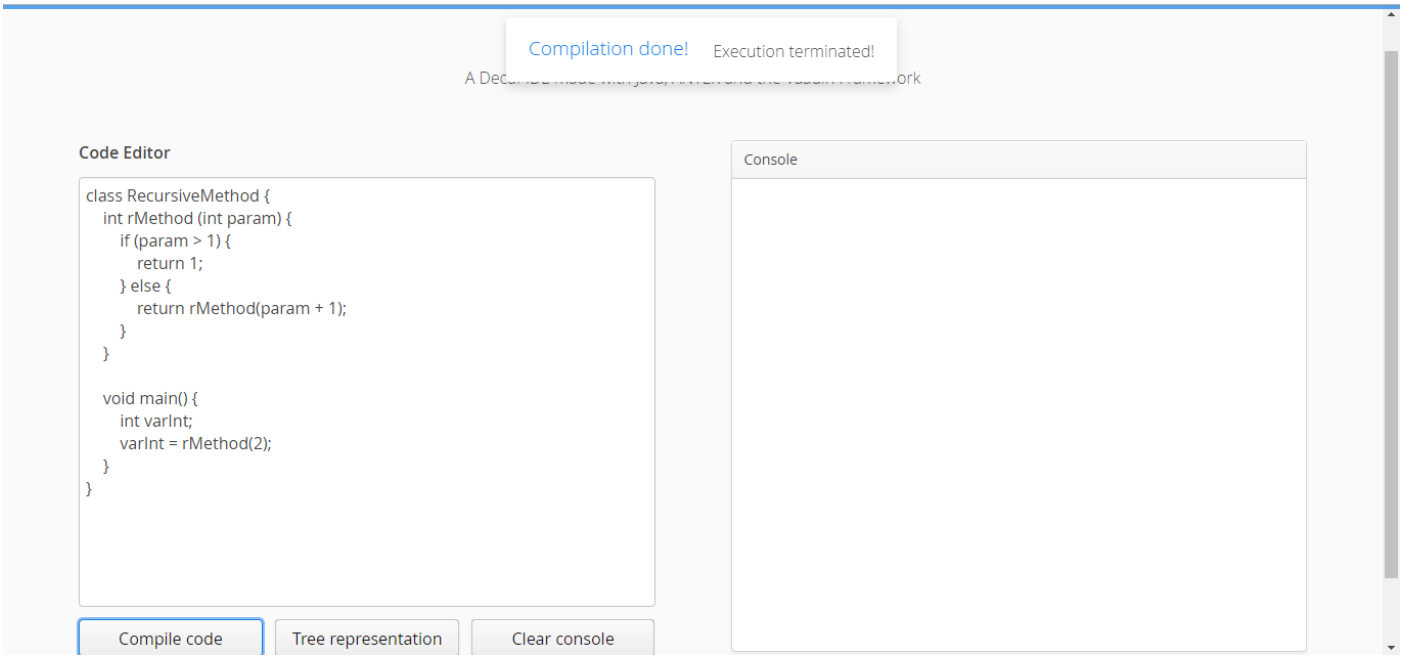


*Figura 4. Ejecución correcta.*

Ahora, una ejecución con errores:

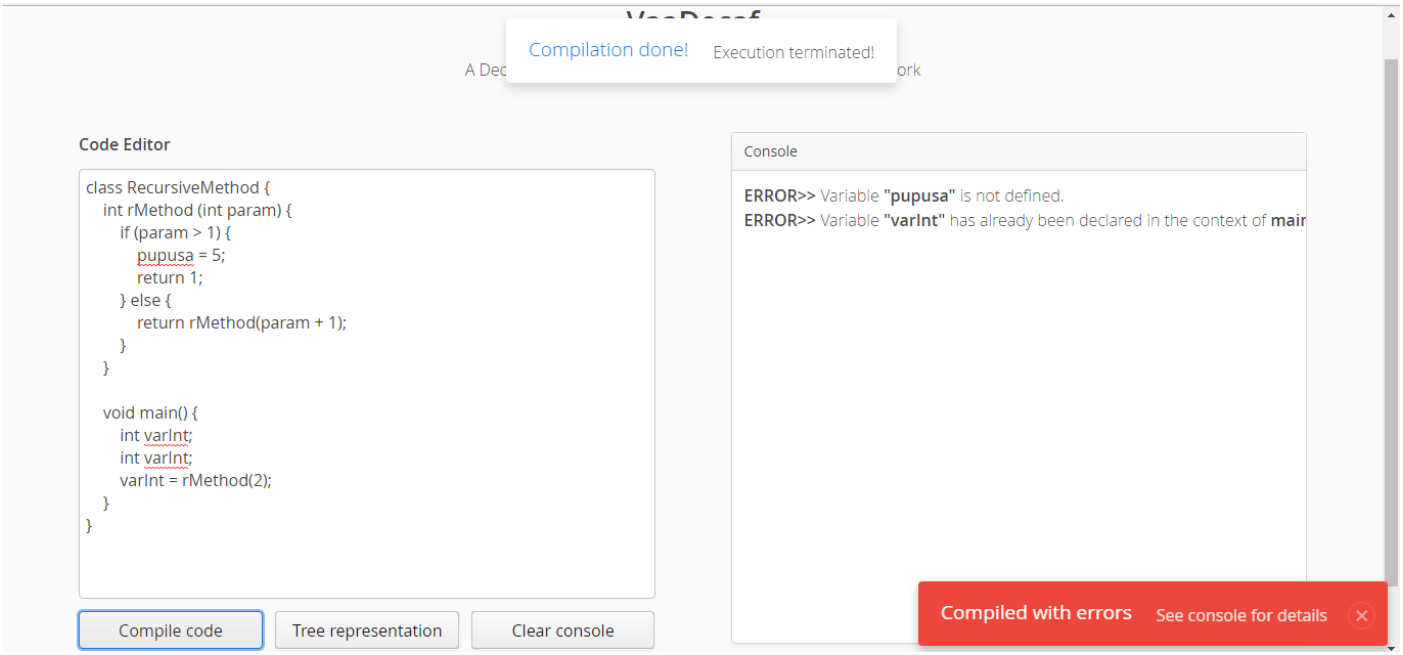


*Figura 5. Ejecución con errores.*

**Pruebas de la generación de código intermedio**

Se emplearon varios archivos de prueba. A continuación, se presenta el código en alto nivel con su respectivo código TAC generado.

```
class TAC1 {
    void main () {
        int a;
        int b;
        int c;
        int d;

        a = b + c + d;
        b = a * a + b * b;
    }
}
```

```
main:
   BeginFunc 20;
   _t0 = b+c;
   a = _t0+d;
   _t1 = a*a;
   _t2 = b*b;
   b = _t1+_t2;
   EndFunc;
```

```
class TAC2 {
    void main () {
        int var1;
        int var2;
        int var3;

        if (var1 < var2) {
            var3 = var1;
        } else {
            var3 = var2;
        }

        var3 = var3 * var3;
    }
}
```

```
main:
   BeginFunc 8;
   _t0 = var1<var2;
   Ifz _t0 Goto _L0;
   var3 = var1;
   Goto _L1;

   _L0:
   var3 = var2;

   _L1:
   var3 = var3*var3;
   EndFunc;
```

```
class TAC3 {
    void main () {
        int x;
        int y;

        while (x < y) {
            x = x * 2;
        }

        y = x;
    }
}
```

```
main:
   BeginFunc 8;

   _L0:
   _t0 = x<y;
   Ifz _t0 Goto _L1;
   x = x*2;
   Goto _L0:

   _L1:
   x = x;
   y = x;
   EndFunc;
```

```
class TAC4 {
    void main () {
        int x;
        int y;
        int m2;
        m2 = x*x + y*y;

        while (m2 > 5) {
            m2 = m2 - x;
        }
    }
}
```

```
main:
   BeginFunc 20;
   _t1 = y*y;
   _t0 = x*x;
   m2 = _t0+_t1;

   _L0:
   _t2 = 5<m2;
   Ifz _t2 Goto _L1;
   m2 = m2-x;
   Goto _L0:

   _L1:
   m2 = m2;
   EndFunc;
```

```
class TAC5 {
    void SimpleFn(int z) {
        int x;
        int y;
        x = x*y*z;
    }

    void main () {
        SimpleFn(137);
    }
}
```

```
_SimpleFn:
   BeginFunc 8;
   _t0 = x*y;
   x = _t0*z;
   EndFunc;
main:
   BeginFunc 8;
   _t0 = 137;
   x = 137;
   PushParam _t0;
   LCall _SimpleFn;
   PopParams 4;
   EndFunc;
```

```
class TAC6 {
    void main () {
        int arr[5];
        arr[0] = 4;
        arr[1] = arr[0] * 2;
    }
}
```

```
main:
   BeginFunc 76;
   _t0 = 0;
   _t1 = 4;
   _t2 = _t0*_t1;
   _t3 = arr+_t2;
   _t4 = *(_t3);
   _t5 = 4;
   _t4 = _t5;
   _t6 = 1;
   _t7 = 4;
   _t8 = _t6*_t7;
   _t9 = arr+_t8;
   _t10 = *(_t9);
   _t12 = 0;
   _t13 = 4;
   _t14 = _t12*_t13;
   _t15 = arr+_t14;
   _t16 = *(_t15);
   _t11 = _t16*2;
```

```
    _t10 = _t11;
    EndFunc;
```

```
class TAC7 {
    void main () {
        int x;
        int y;
        boolean b;

        b = x <= y;
    }
}
```

```
main:
    BeginFunc 12;
    _t0 = x<y;
    _t1 = x==y;
    b = _t0||_t1;
    EndFunc;
```

```
class TAC8 {
    int foo(int a, int b) {
        return a + b;
    }

    void main () {
        int c;
        int d;

        foo(c,d);
    }
}
```

```
_foo:
    BeginFunc 8;
    _t0 = a+b;
    _t1 = a+b;
    Return _t0;
    EndFunc;
main:
    BeginFunc 16;
    _t0 = d;
    _t1 = c;
    PushParam _t0;
    PushParam _t1;
    LCall _foo;
    PopParams 8;
    EndFunc;
```

```
class TAC9 {
    struct Dog {
        int id;
        struct Owner owner;
    }

    void main() {
        struct Dog dog;
        dog.id = 1;
    }
}
```

```
main:
    BeginFunc 28;
    SCall dog.id
    _t0 = 0;
    _t1 = 4;
    _t2 = _t0*_t1;
    _t3 = dog+_t2;
    _t4 = *(_t3);
    _t5 = 1;
    _t4 = _t5;
    End SCall;
    EndFunc;
```

**Pruebas de la generación de código MIPS**

program.decaf

Programa en decaf

```
class Factorial {

  int factorial(int n) {
    int ans;
    int i;

    ans = 1;
    i = n;

    while (i > 1) {
        ans = ans * i;
        i = i-1;
    }

    print(ans);
    return ans;

  }

  void main() {
    factorial(5);
  }

}
```

decaf.tac

Código intermedio

```
_factorial:
   BeginFunc 16;
   ans = 1;
   i = n;

   StartWhile:

   _L0:
   _t0 = 1<i;
   Ifz _t0 Goto _L1;
   ans = ans*i;
   i = i-1;
   Goto _L0:

   _L1:
   PushParam ans;
   LCall print;
   PopParams 1;
   _t1 = ans;
   _t2 = ans;
   Return _t1;
   EndFunc;
main:
   BeginFunc 8;
   _t0 = 5;
   PushParam _t0;
   LCall _factorial;
   PopParams 4;
   EndFunc;
```

generated_mips.asm

Código en MIPS

```
# auto generated MIPS file.
# VaaDecaf, a Decaf compiler made with Java, ANTLR
and the Vaadin framework


.text
.globl main
main:
   li $t0, 5
   move $a0, $t0
   sw $a0, n_factorial
   jal factorial
   move $s0, $v0


   # ---------- Exit ----------
   li $v0, 10
   syscall


factorial:
   addi $sp, $sp, -4           # Adjust stack
pointer
   sw $s0, 0($sp)              # Save reg
   addi $sp, $sp, -4           # Adjust stack
pointer
   sw $s1, 0($sp)              # Save reg
   addi $sp, $sp, -4           # Adjust stack
pointer
   sw $s2, 0($sp)              # Save reg
   addi $sp, $sp, -4           # Adjust stack
pointer
   sw $s3, 0($sp)              # Save reg
   li $s0, 1
   sw $s0, ans_factorial            # str data
   lw $t0, n_factorial              # ld data
n_factorial
   move $s1, $t0
   sw $s1, i_factorial              # str data
   _L0:
   li $t1, 1
   lw $t2, i_factorial              # ld data
i_factorial
   slt $t0, $t1, $t2
   blez $t0, _L1
   lw $t2, ans_factorial            # ld data
ans_factorial
   lw $t1, i_factorial              # ld data
i_factorial
   mult $t2, $t1
   mflo $s2
   sw $s2, ans_factorial            # str data
   lw $t1, i_factorial              # ld data
i_factorial
   li $t2, 1
   sub $s3, $t1, $t2
   sw $s3, i_factorial              # str data
   b _L0
   _L1:
   li $v0, 1              # Print call
   lw $a0, ans_factorial
   syscall
   lw $t0, ans_factorial            # ld data
ans_factorial
   move $t1, $t0
   lw $t3, ans_factorial            # ld data
ans_factorial
   move $t2, $t3
   move $v0, $t1              # Return
   lw $s4, 0($sp)             # Restore reg
   addi $sp, $sp, 4           # Adjust stack
pointer
   lw $s3, 0($sp)             # Restore reg
   addi $sp, $sp, 4           # Adjust stack
pointer
   lw $s2, 0($sp)             # Restore reg
   addi $sp, $sp, 4           # Adjust stack
pointer
   lw $s1, 0($sp)             # Restore reg
```

```
class TAC2 {
    void main () {
        int var1;
        int var2;
        int var3;

        if (var1 < var2)
{
            var3 = var1;
        } else {
            var3 = var2;
        }

        var3 = var3 *
var3;
    }
}
```

```
main:
   BeginFunc 8;
   _t0 = var1<var2;
   Ifz _t0 Goto _L0;
   var3 = var1;
   Goto _L1;
   _L0:
   var3 = var2;

   _L1:
   var3 = var3*var3;
   EndFunc;
```

```
   addi $sp, $sp, 4            # Adjust stack
pointer
   jr $ra                   # Jump to addr stored in
$ra


# ---------- data section ----------
.data
n_factorial:            .word 0
ans_factorial:          .word 0
i_factorial:            .word 0
```

```
# auto generated MIPS file.
# VaaDecaf, a Decaf compiler made with Java, ANTLR
and the Vaadin framework


.text
.globl main
main:
   lw $t1, var1_main           # ld data var1_main
   lw $t2, var2_main           # ld data var2_main
   slt $t0, $t1, $t2
   blez $t0, _L0
   lw $t2, var1_main           # ld data var1_main
   move $s0, $t2
   sw $s0, var3_main           # str data
   b _L1
   _L0:
   lw $t2, var2_main           # ld data var2_main
   move $s1, $t2
   sw $s1, var3_main           # str data
   _L1:
   lw $t2, var3_main           # ld data var3_main
   lw $t1, var3_main           # ld data var3_main
   mult $t2, $t1
   mflo $s2
   sw $s2, var3_main           # str data


   # ---------- Exit ----------
   li $v0, 10
   syscall


# ---------- data section ----------
.data
var1_main:          .word 0
var2_main:          .word 0
var3_main:          .word 0
```

**Repositorio**

El repositorio **privado** del proyecto se encuentra en: https://github.com/gbrolo/decaf-ide

**Diagrama de Clases UML**

Se adjunta un diagrama de clases en el directorio del proyecto, en la ruta: *src/main/java/com/brolius/UML/class-diagram.png* En dicho archivo se puede visualizar de mejor forma el diagrama, el cual es el mismo presentado a continuación: