Proyecto: Compilador de Lenguaje C-TDS

Asignatura: Taller de Diseño de Software

2016

Integrantes: Leandro Buttignol, Martín Lopez, Mariano Coria

Docentes: Pancho y Ceci

***Introducción***

En el presente informe se han de detallar las decisiones de diseño tomadas en cada una de las etapas de construcción del compilador para el lenguaje C-TDS.

Para conocer en profundidad las etapas y el lenguaje en sí, remitirse a los archivos complementarios a este informe, los mismos son citados a continuación.

**00-TDS-proyecto.pdf.** En este archivo se encuentra la descripción del proyecto.

**01-TDS-spec-lenguaje.pdf.** En este archivo se encuentra la especificación del lenguaje para el cual se quiere implementar un compilador.

***Primera Etapa: Análisis Léxico y Sintáctico***

**Herramientas de desarrollo utilizadas**

* + JFlex
  + Sublime text 3
  + Ant
  + CUP
  + Eclipse

**Objetivos de desarrollo en la primera etapa**

Los objetivos parciales concernientes al desarrollo del compilador en esta etapa temprana son la implementación de scanner y parser, como así también la definición de la gramática.

**Grammar/Parser**

En esta primera etapa se busca corroborar que el código escrito en un archivo sea una cadena correspondiente al lenguaje C-TDS definido en la cátedra, es decir, discriminar las palabras reservadas del lenguaje.

En primer instancia, se realizó la implementación del scanner, para ello se utilizó la herramienta JFlex.

Una vez que dicha implementación del scanner se consideró terminada, se prosiguió con la implementación del parser utilizando la herramienta CUP. Es aquí donde se define la gramática de nuestro lenguaje C-TDS.

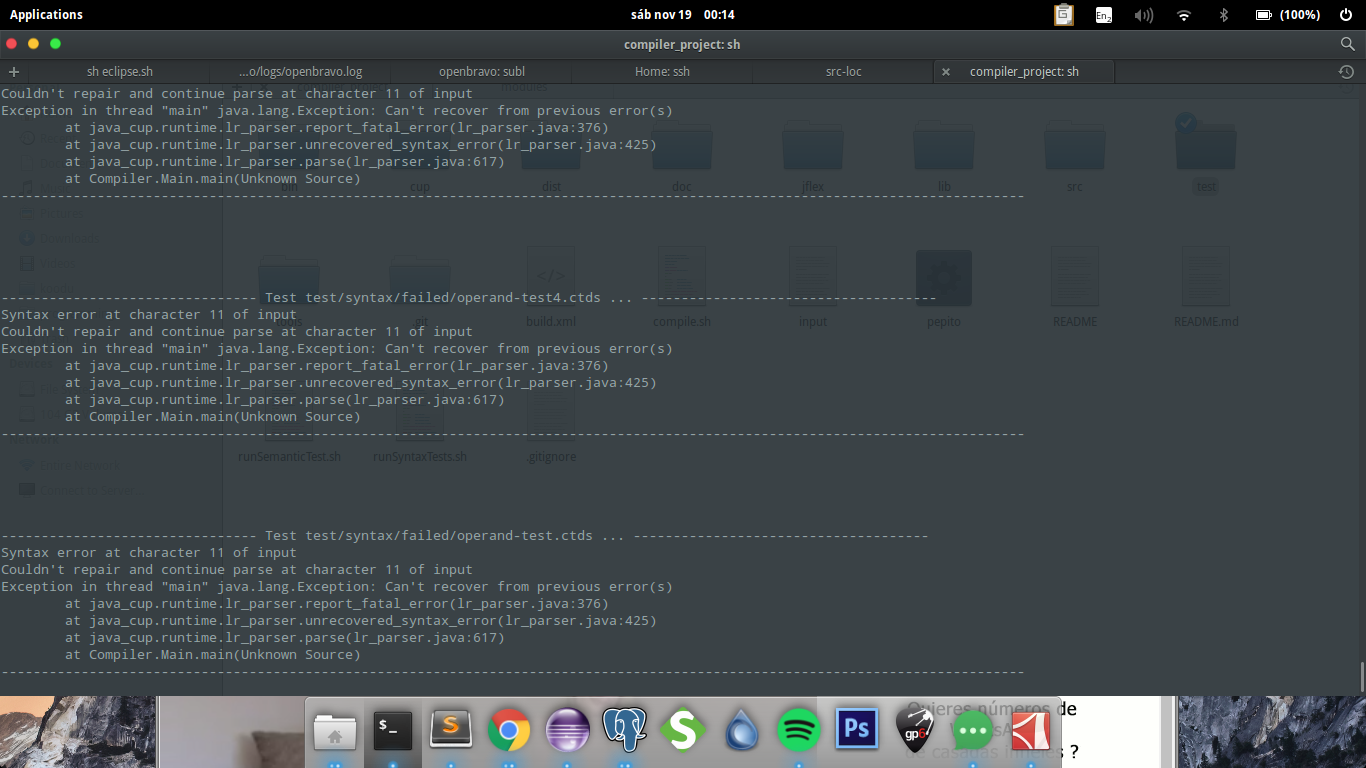
**Problemas**

Uno de los problemas enfrentados durante la ocurrencia de esta primera etapa fue el reconocimiento de multiples comentarios multilinea (el cual pudo ser solucionado). La inclusión de facilidades para el programador como el aumento y decremento automático, las mismas fueron causantes de complicaciones en la implementación por lo que se decidió no incluirlas en este entrega.

**Testing**

Se realizó una batería de numerosos tests con distintos niveles de complejidad y rigurosidad, en los mismos se realizaron casos de prueba para todas las sentencias que se pueden declarer en nuestro lenguaje, como así también para todos los tipos de variables y operaciones entre ellas.

Todos los errores de sintaxis son manejados por el parser, a continuación algunos ejemplos. Estos son resultados obtenidos en la batería de tests.



***Segunda etapa: Análisis Semántico***

**Introducción/Objetivo**

Al hablar de un compilador resulta inevitable la inherente conexión a los chequeos semánticos que determinen a priori la corrección del código antes de proseguir a generar la traducción a assembler o el respectivo binario ejecutable. En esta etapa, nos dedicamos íntegramente a estos chequeos, los mismos pueden dividirse en 2 grandes ramas que son alcance y coherencia de tipos.

Como estructura para soportar el programa como objeto se implementó una jerarquía de clases que abarcaba desde lo más general que era un programa, hasta lo más pequeño, que en nuestro caso son los literales.

***Alcance***

Cada vez que se invoque a una variable o método, éste debe estar definido de tal manera que, respetando las convenciones del lenguaje, sea alcanzable desde el bloque de código en la cual se lo invoca. Como convención para ctds se adopta el sistema de alcance de java, que posee un alcance estático **-*static scope-*.** Por lo tanto, cada vez que se invoque a un identificador, éste ha de estar definido en el mismo nivel o uno superior y siempre respetando el encapsulamiento de información, es decir, en un nivel superior pero dentro de la misma clase.

Para llevar a cabo los chequeos de alcanzabilidad, se definió una clase visitor denominada BuilderVisitor, esta tiene entre sus atributos una instancia de la clase SymbolTable, que no es más que una pila, en la cual cada nivel es una lista de definiciones de símbolos, existiendo un nivel por cada bloque anidado. En concreto, cada vez que el builder visita a un bloque, genera un nuevo nivel en esa pila con una lista de todos los indentificadores que se encuentren definidos en el bloque en cuestión. Estos niveles se eliminan al terminar de recorrer el bloque. Entonces, cuando se invoca a un identificador se lo busca en la lista de símbolos del nivel actual, en caso de no encontrarlo se lo busca en el nivel inmediato inferior hasta encontrarlo o llegar al nivel cero (en este caso, el identificador no está definido y estamos frente a un error semántico). A su vez, cada vez que se define un identificador se chequea que no exista otro con el mismo nombre en el nivel actual, evitando así posibles ambigüedades.

**Coherencia de tipos**

La coherencia de tipos hace referencia al control requerido por un lenguaje en particular en términos de castings posibles. Es decir, cada valor (y por lo tanto cada identificador) está relacionado inequívocamente a un tipo, y éste, restringe las operaciones que pueden ser ejecutadas sobre el identificador mismo (dichas restricciones han de ser inexorablemente respetadas). Así también, cada expresión (que puede estar conformada por numerosas operaciones entre valores/identificadores) está relacionada a un tipo y las operaciones que se le apliquen deben adecuarse a este.

La implementación de estos controles siguió una metodología similar a la utilizada en los controles de alcance, en este caso el Visitor es el TypeCheckVisitor. Luego de chequear que todos los identificadores tengan un alcance apropiado, debemos cerciorarnos de que ese valor con el que se relaciona al identificador sea del tipo apropiado. También utilizamos una tabla de símbolos, en la cual buscamos por medio del identificador invocado la primer ocurrencia de una declaración de dicho identificador en el stack, esta debe ser del mismo tipo que el usado en la invocación. Cuando una expresión no es literal no tiene el tipo seteado a priori, por lo tanto, antes de realizar cualquier chequeo se verifica si el tipo de la expresión está o no definido. En caso de que el tipo no esté definido, se ejecuta un visit sobre esa expresión, este visit recorre los componentes de la expresión de manera recursiva (el caso base es un identificador o un literal, cuyo tipo siempre está seteado, teniendo esto el back tracking setea los tipos a cada expresión a la que este pertenece –cada expresión puede estar conformada por muchas expresiones-).

**Problemas/Soluciones**

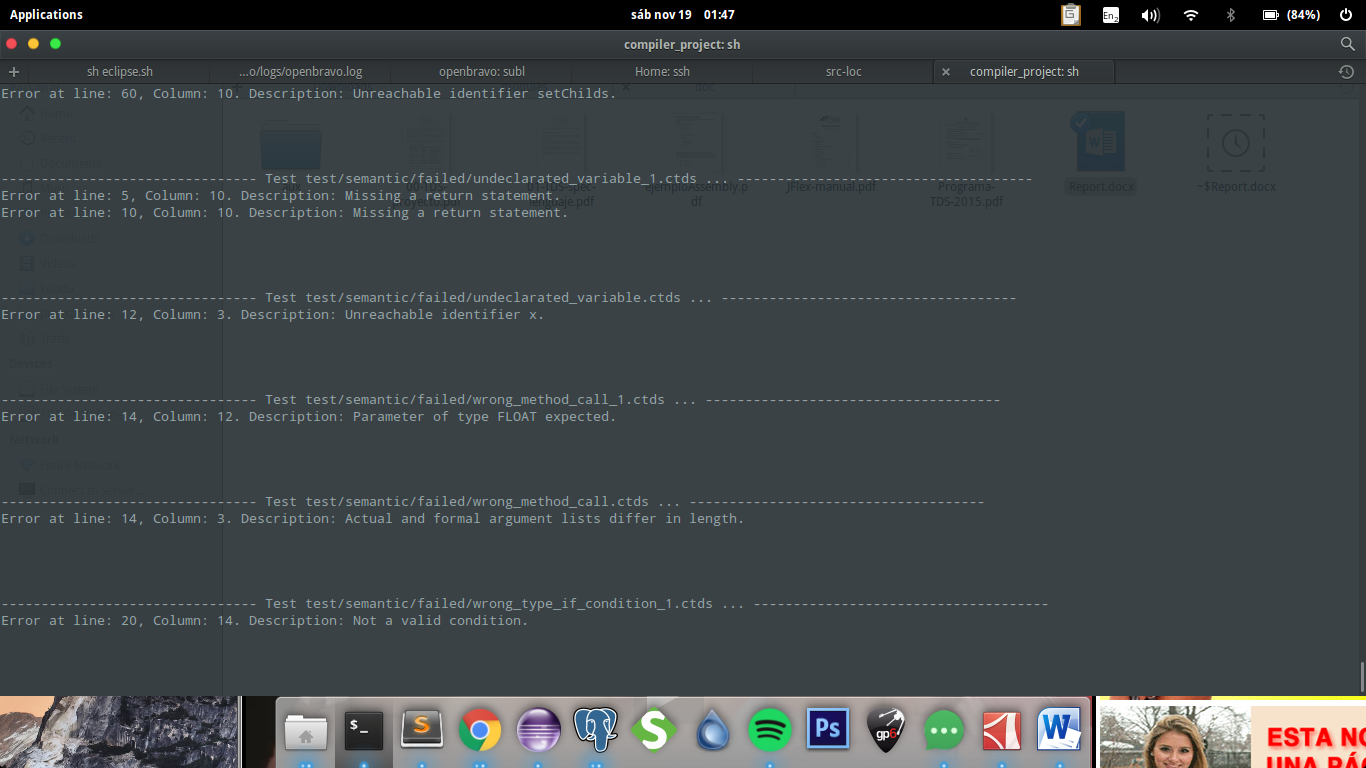
En primera instancia se deseaba agregar las funcionalidades “++” y “--” que no estaban en las especificaciones del lenguaje original, pero debido a un problema de ambigüedad que existía entre el operando “--” (auto-decremento) y la doble negación aritmética se decidió no agregar este par de operaciones.

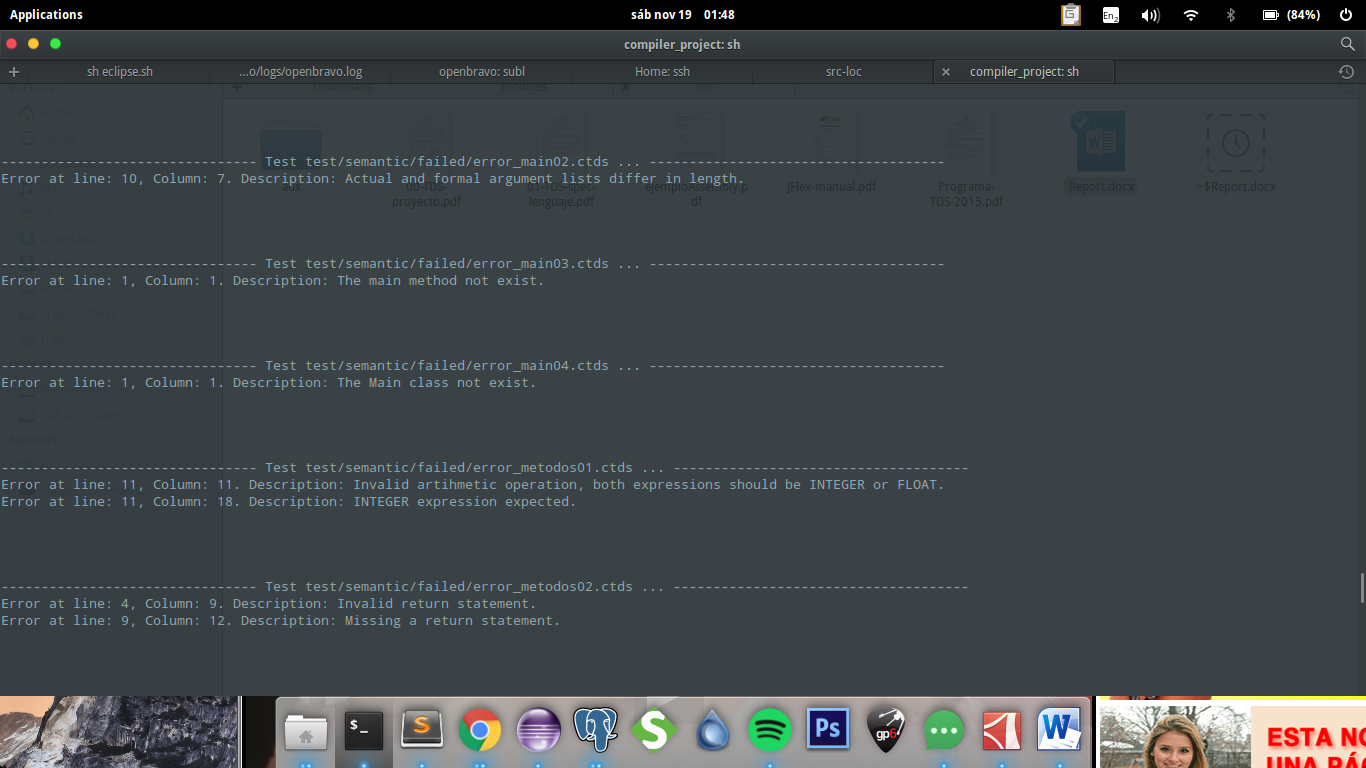
A medida que se avanzó en el análisis semántico se vio la necesidad de retornar errores adecuadamente (con una línea y una columna) y tuvimos que modificar completamente nuestro AST debido a que en ningún constructor habíamos definido estos valores.

**Testing**

AL igual que en la etapa anterior se realizó una *test suit* específica para controlar el análisis semántico del compilador, esta *test suit* posee tanto test para éxito como para falla. Para ejecutarlos y controlar el resultado de la *test suit* se programó un script que lo realizaba automáticamente.

A continuación algunos ejemplos de los errores que se verifican en esta etapa, como identificadores cuya declaración es inalcanzable, errores en tipos, sentencia return faltantes y demás.





***Tercera Etapa: Código intermedio***

**Introducción/Objetivo**

En esta etapa se busca obtener una expresión lógica equivalente al programa fuente, la misma puede verse como una representación más cercana a assembler, hecha mediante un sistema de tres direcciones. Cada statement es representado mediante 3 variables de tipo Operand, con estos pudiendo ser cualquier tipo de expresión ya sean valores temporales, invocaciones a variables o métodos, etc. El resultado es una secuencia de listas de instrucciones representadas mediante 3 operandos cada una, abstrayéndonos así de las diferencias entre el código fuente y el assembler generado, pues obtenemos una representación cuya traducción resulta mucho más sencilla.

**Problemas/Soluciones**

Un problema de esta etapa fue construir el set de instrucciones que iba a soportar el lenguaje de código intermedio, y darle la semántica adecuada. Para resolverlo se decidió ir agregando sentencias a medida que se requerían nuevas funcionalidades no soportadas.

Algo que no se pudo resolver hasta la etapa siguiente (Generación de assembler) fue el acceso a atributos de la clase y a arreglos, en un principio se generaba un label que indicaba que se estaba por crear o acceder a uno de estos elementos, cuando llegamos a general assembler vimos que la creación del label no servía de nada y que en realidad el tratamiento de estos elementos era similar al de una expresión a evaluar, es decir se debía acceder y almacenar el resultado en un temporal.

**Testing**

En un principio las pruebas eran realizadas mentalmente, es decir comparando el código fuente con el intermedio generado. Para facilitar la labor de testing, implementamos un *pretty print* que muestra a modo de texto la representación lógica en código de tres direcciones.

***Cuarta Etapa: Generación de Assembler***

**Introducción/Objetivo**

Aquí se toma el código de tres direcciones generado en la etapa anterior y a partir del mismo se genera un código assembler equivalente, en nuestro caso, elegimos traducir a GNU assembler (64 bits). A diferencia de las etapas anteriores, no utilizamos un patrón visitor, ya que la entrada de datos es una lista de instrucciones en vez de una estructura no lineal, como sí lo es el árbol generado por el parser.

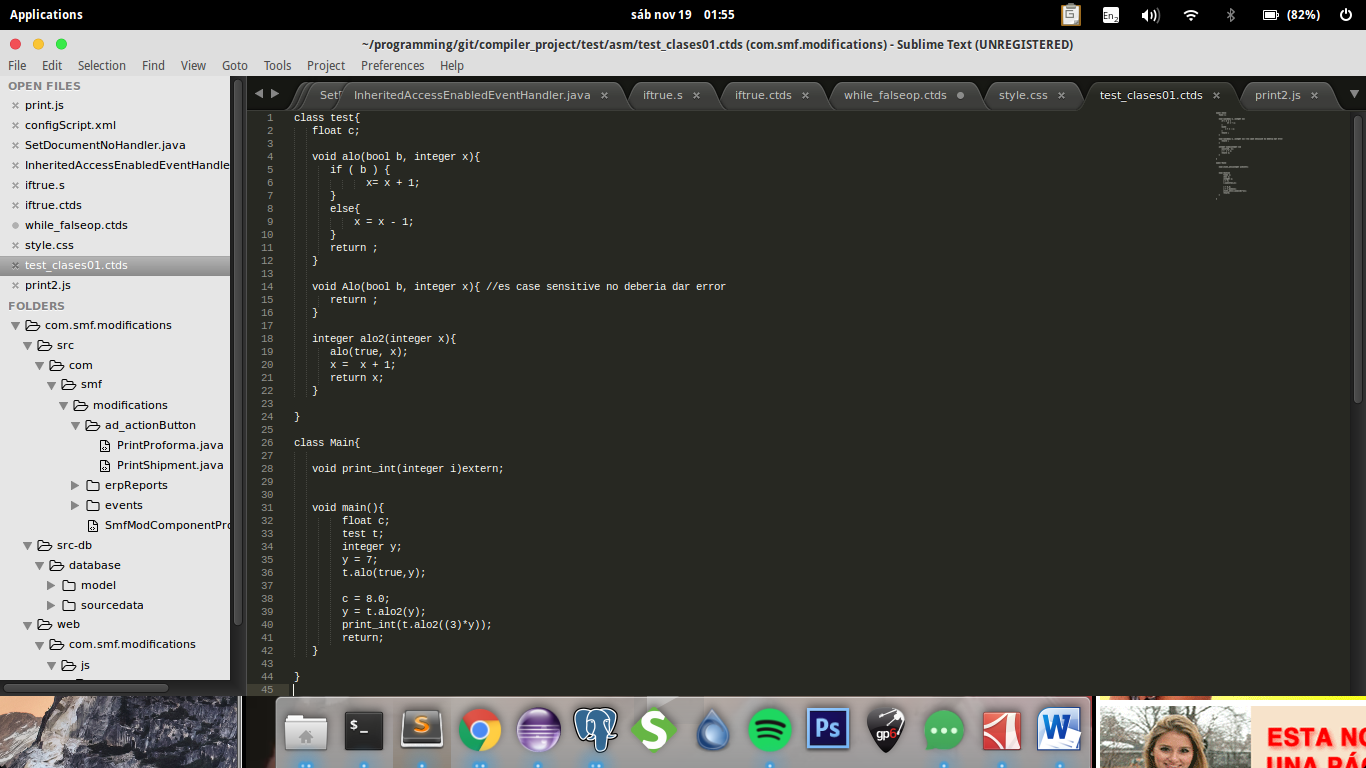
**Problemas/Soluciones**

El principal problema con el que nos encontramos en esta etapa referida a la generación de assembler, fue la insuficiente determinación de offsets para las expresiones involucradas. Esto se debió en gran medida a la falta de previsión en las etapas anteriores, lo solucionamos volviendo al código de etapas anteriores y agregando ahí toda la información que faltaba, modificando y haciendo ajustes por doquier. La determinación de los offsets de temporales se implementó en el Generador de código intermedio, y, todos los demás offsets precisados se determinaron en el TypeCheckVisitor.

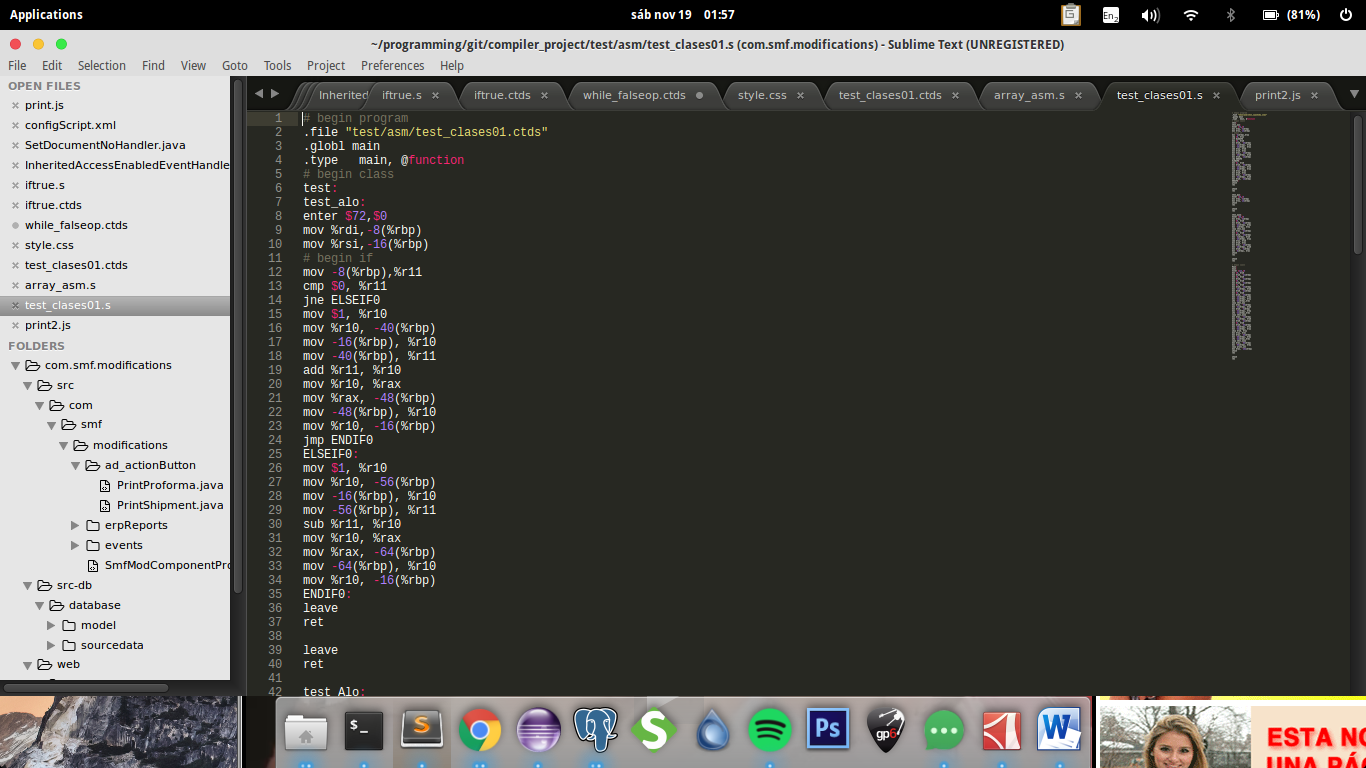
**Testing**

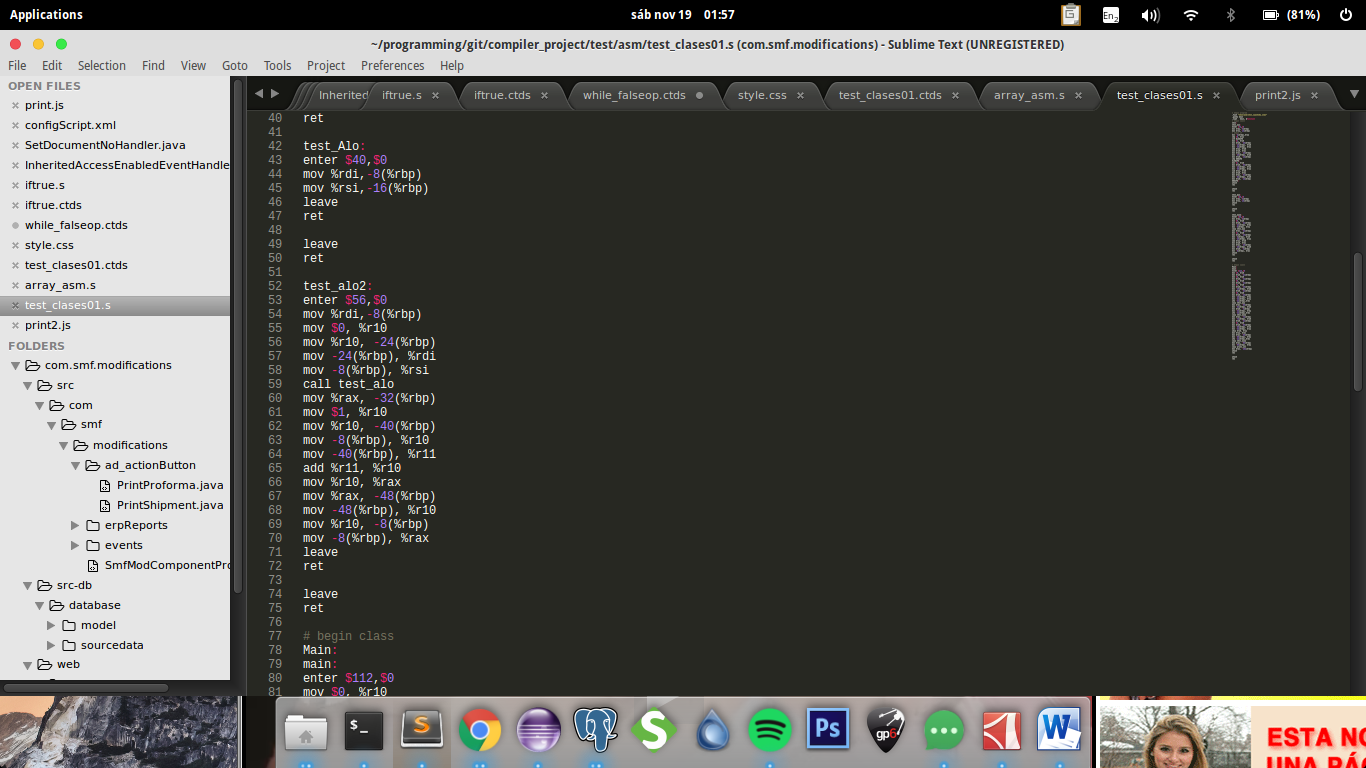
Se realizó una batería de numerosos tests, lo mismos consisten en pequeños programas escritos en lenguaje ctds. Cada test estuvo destinado a probar la corrección de uno o más casos particulares de traducción a assembler, tales como operaciones aritméticas unarias y binarias; llamadas a métodos; sentencias if, for, while; etc. Se definió un conjunto de funciones externas para mostrar por pantalla los valores que se consideraban críticos para cada test.

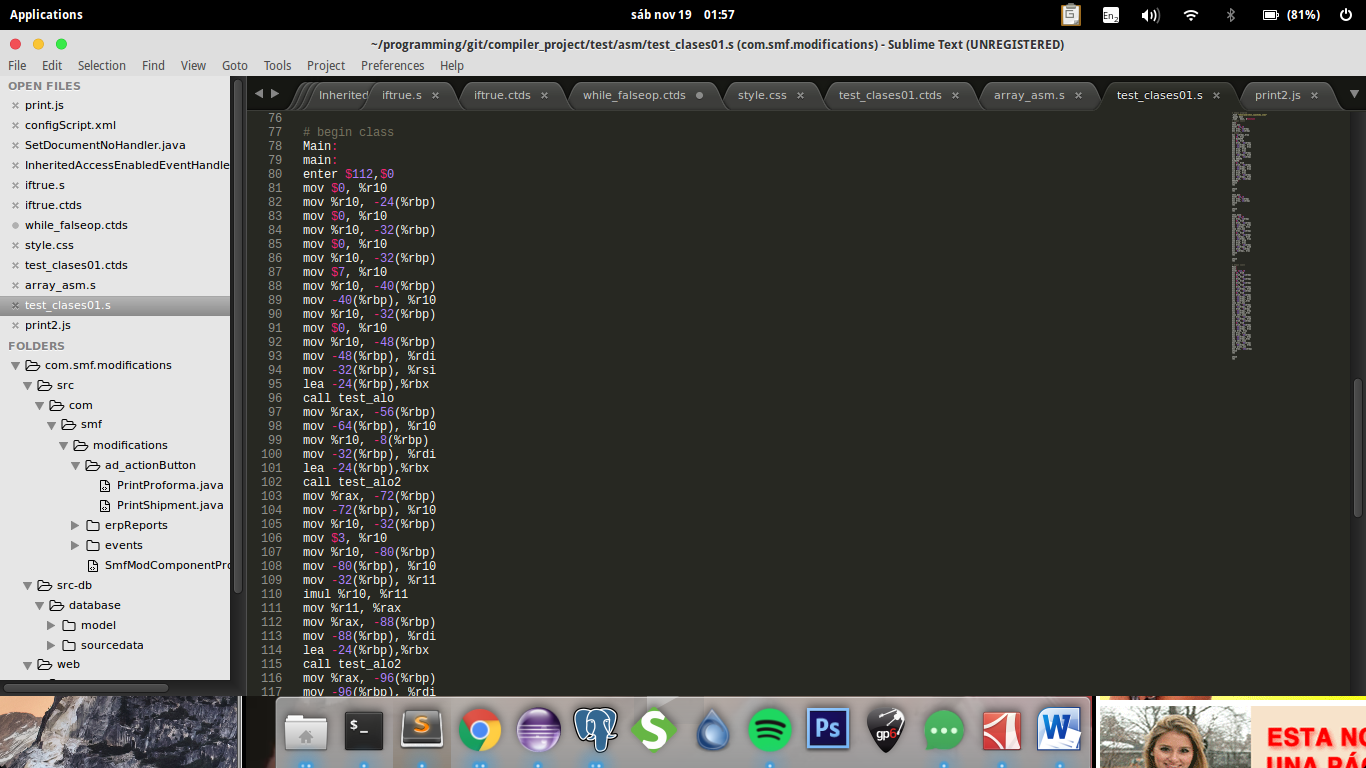
Estos son algunos ejemplos de los tests que se corrieron, el primero para probar el manejo de clases.

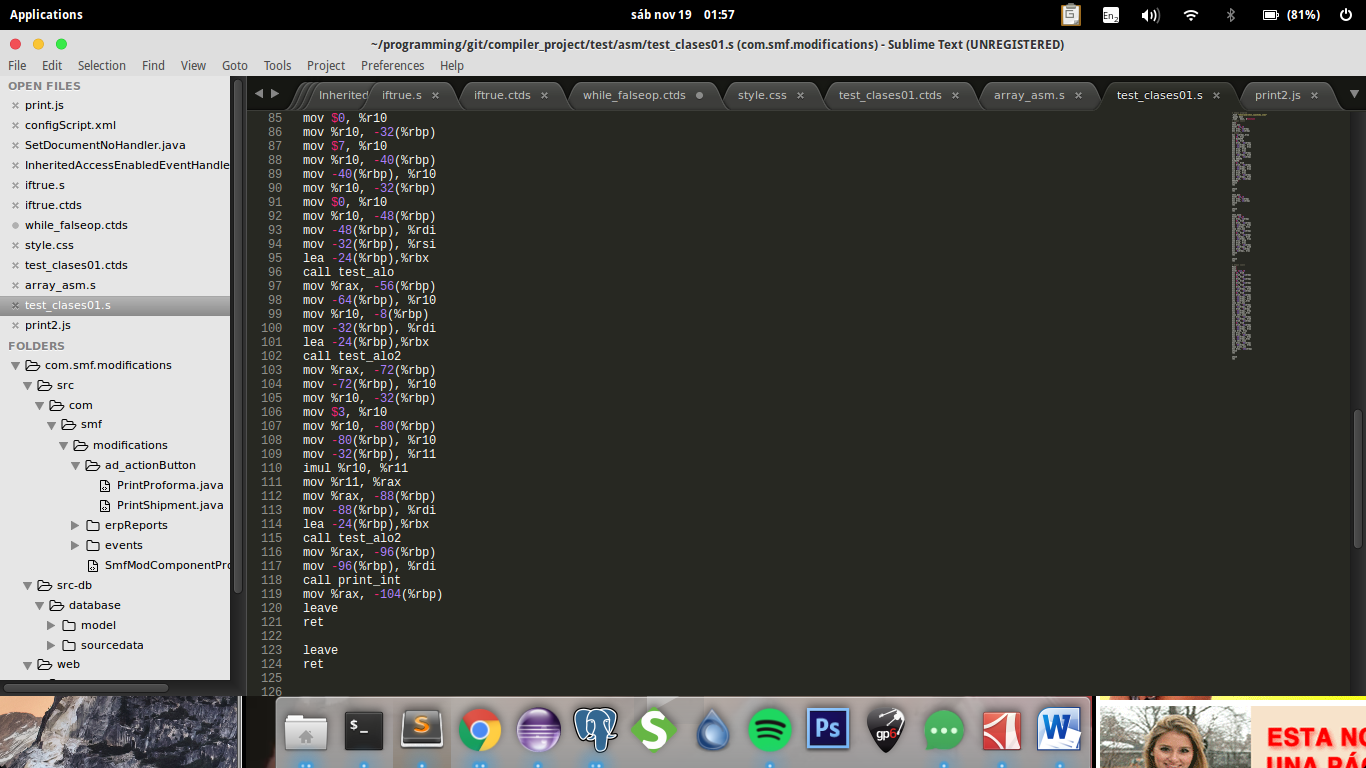
******

Y aqui el assembler obtenido

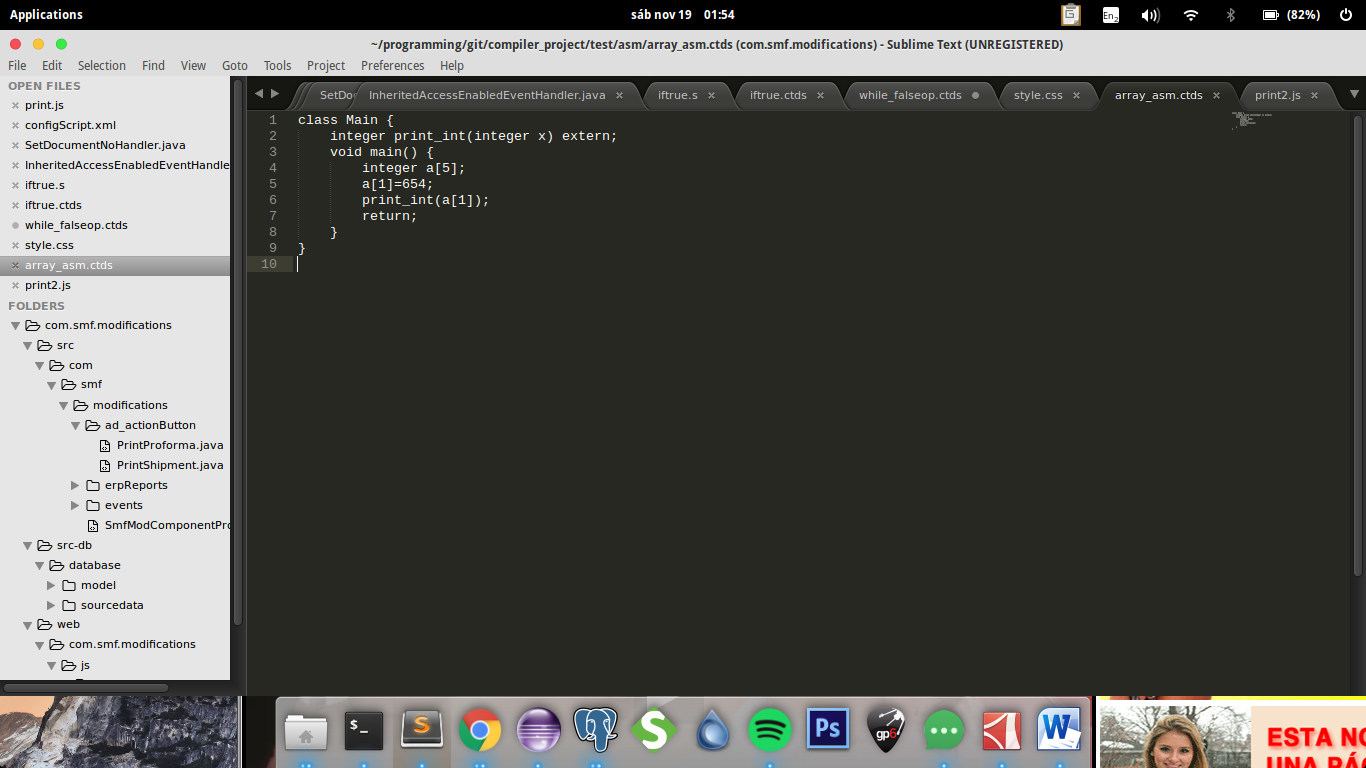
******

******

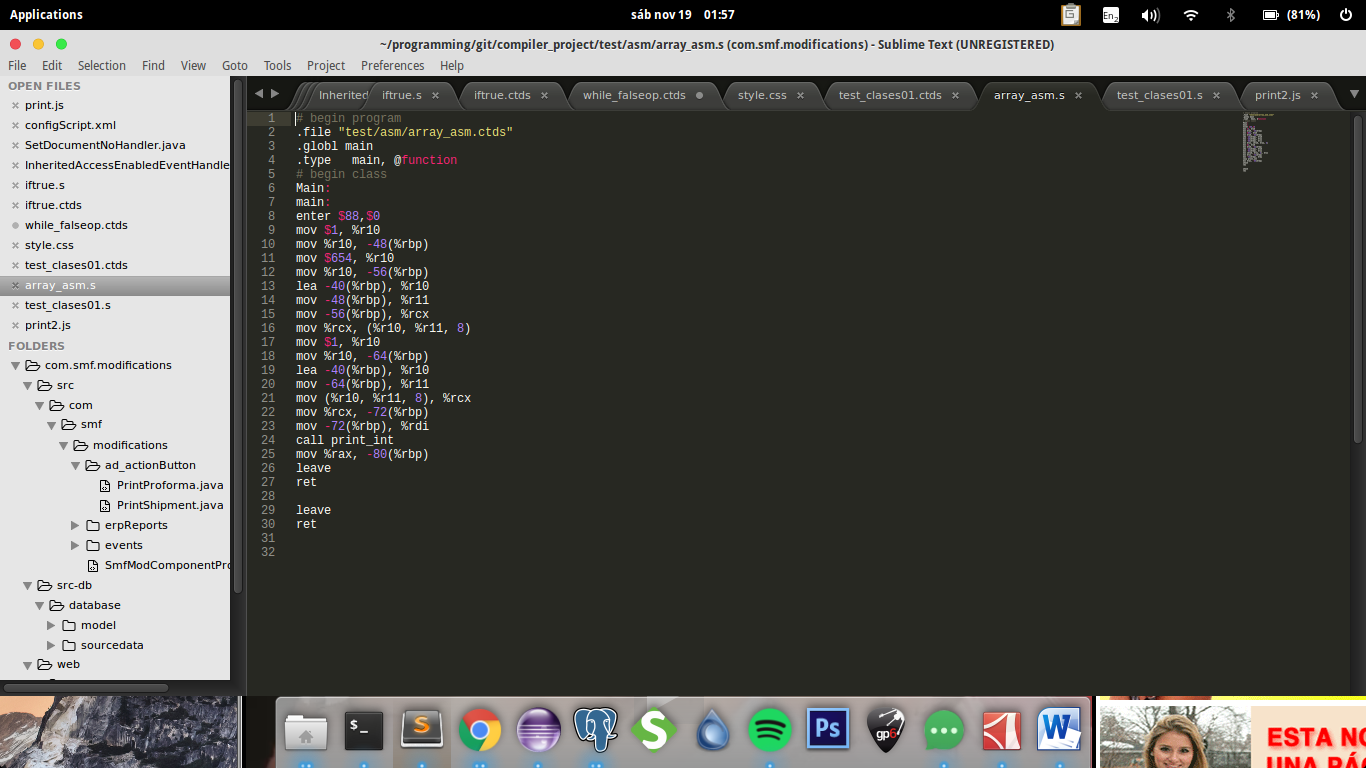
******

******

Otro ejemplo, esta vez probando manipulación de arreglos.

******

Y su respectivo assembler generado.

******

***Quinta Etapa: Optimizaciones***

**Introducción/Objetivo**

En esta etapa se implementaron dos tipos de optimizaciones, las mismas fueron desarrolladas mediante un Visitor que se ejecuta opcionalmente antes de generar el código intermedio.

**Propagación de constantes**

La idea de esta optimización es ahorrar líneas de código assembler obviando operaciones innecesarias, en particular, aquellos cálculos realizados entre literales de manera exclusiva. Es decir, aquellas expresiones que a pesar de estar representados por operaciones son constantes, y por lo tanto, no es requerido que se calculen en cada ejecución del programa. Se implementó esta optimización para expresiones conformadas íntegramente por booleanos y/o enteros literales.

**Eliminación de Código muerto**

En el mismo visitor que se realiza la propagación de constantes se puede buscar y eliminar código inalcanzable (ergo, inútil).

Los diversos casos son los citados a continuación:

* **While false:** no es necesario incluir el código del bloque ya que nunca será ejecutado.
* **IfThenElse con condición false:** no es necesario incluir el código del Then.
* **IfThenElse con condición true:** análogo a IfThenElse con condición false.
* **IfThen con condición false:** no es necesario incluir nada de esta sentencia en absoluto.
* **Break, Continue, Return:** No es necesario incluir las líneas de código ejecutadas de manera secuencial después de alguna de estas sentencias.

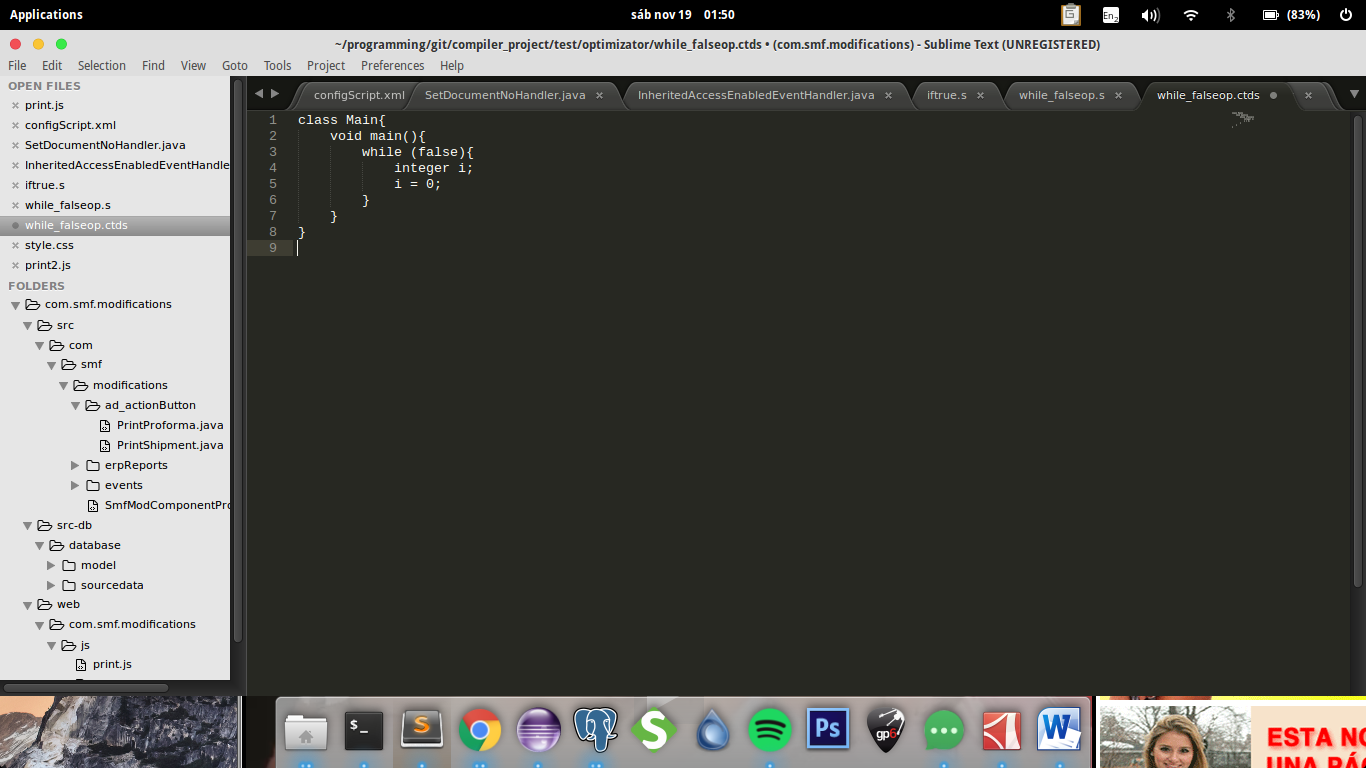
**Problemas/Soluciones**

No nos encontramos con problemas mayores en esta etapa.

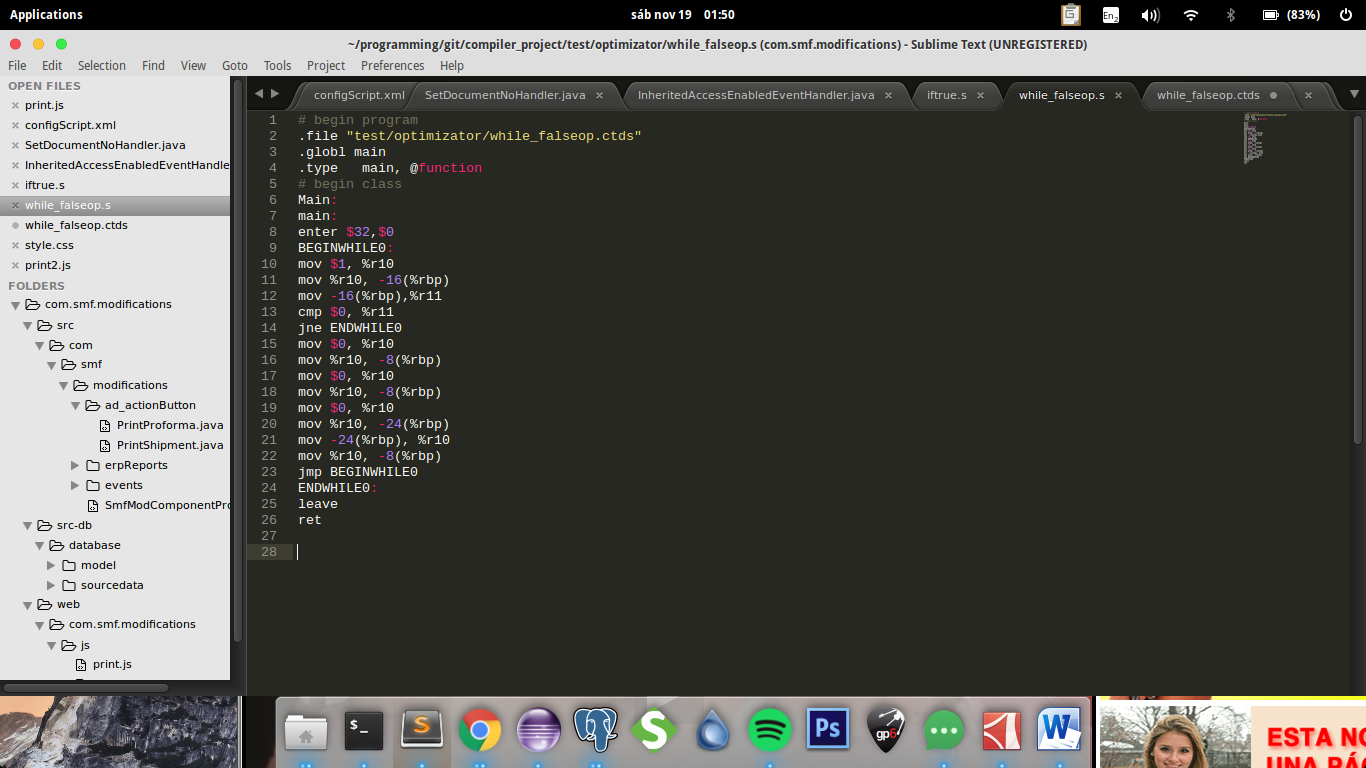
**Testing**

Para probar esta etapa se generaron diversos casos de test para cada escenario y se analizó el código al compilar con optimización y sin optimización.

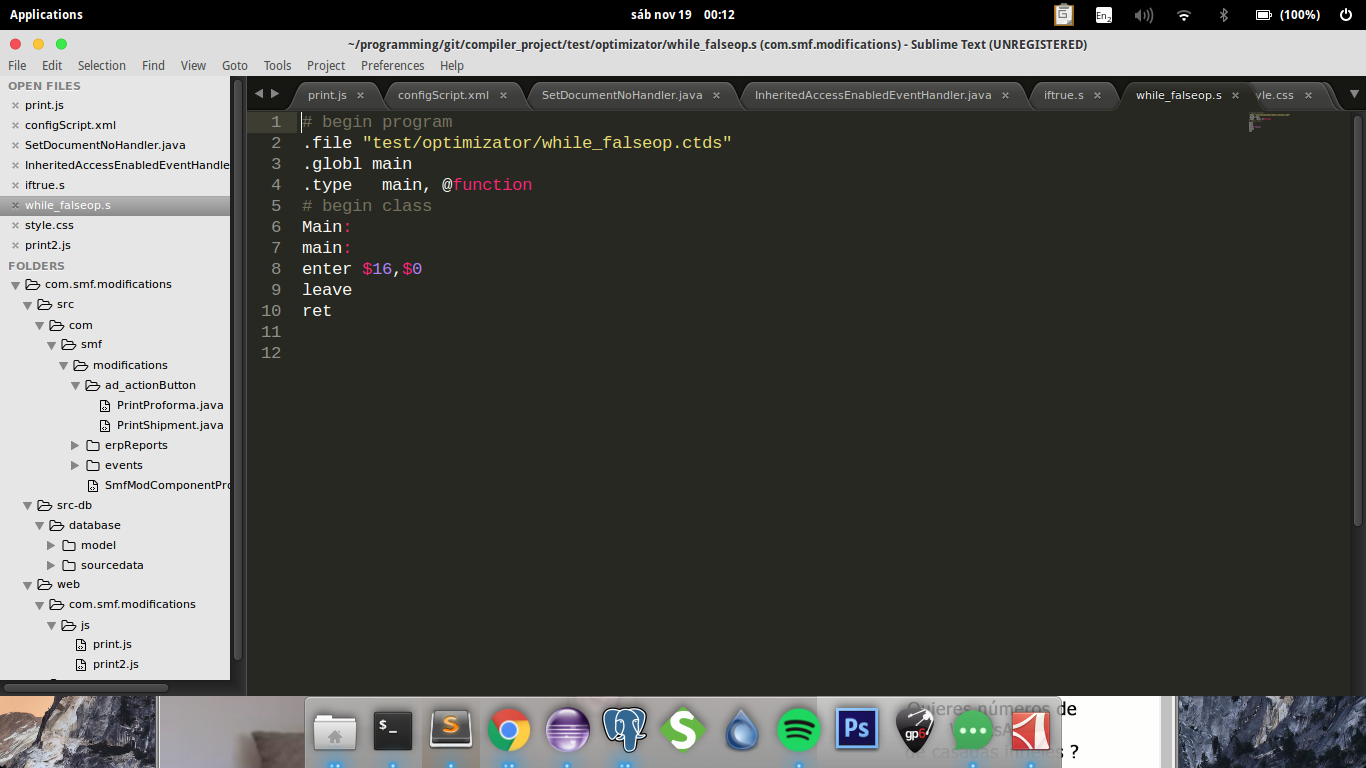
Este es un ejemplo de los tests realizados, en este caso en particular un while(false).

******

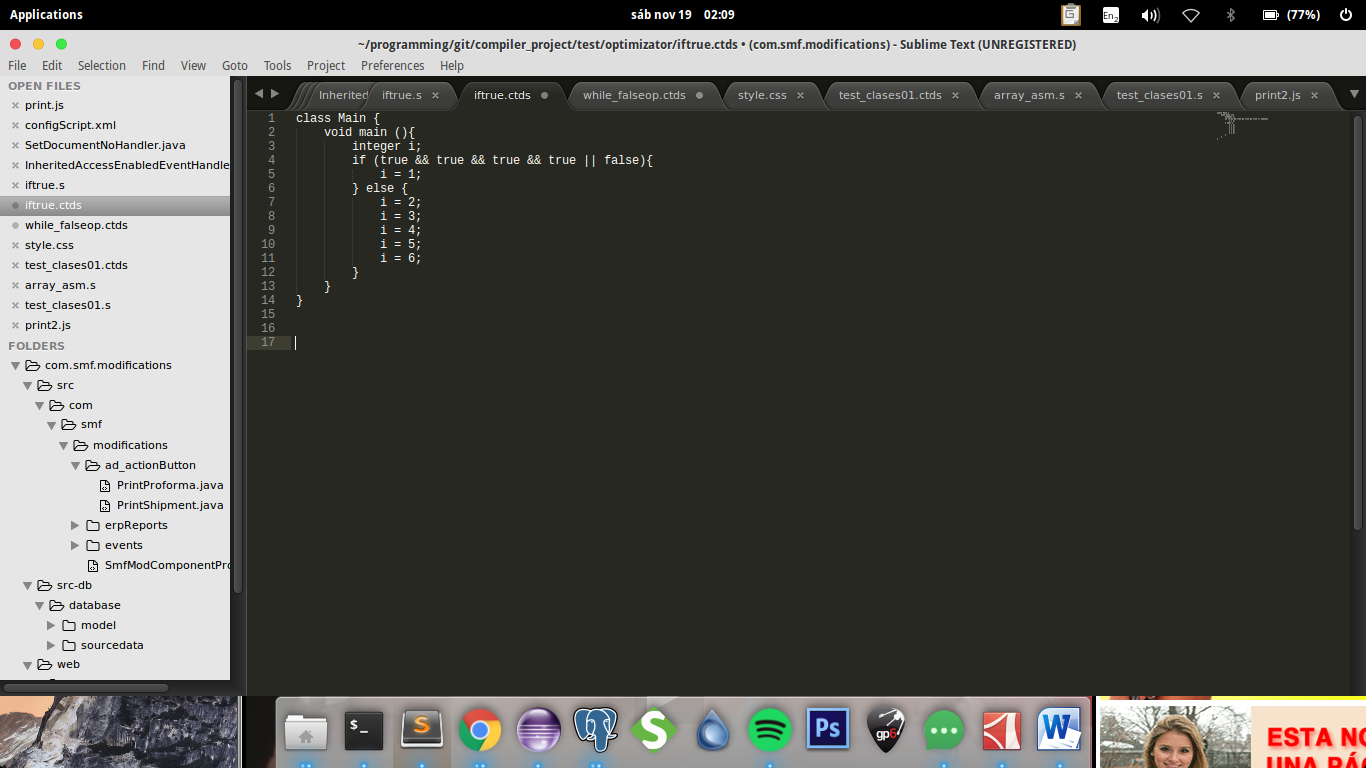
Este es el código assembler sin optimizaciones.

******

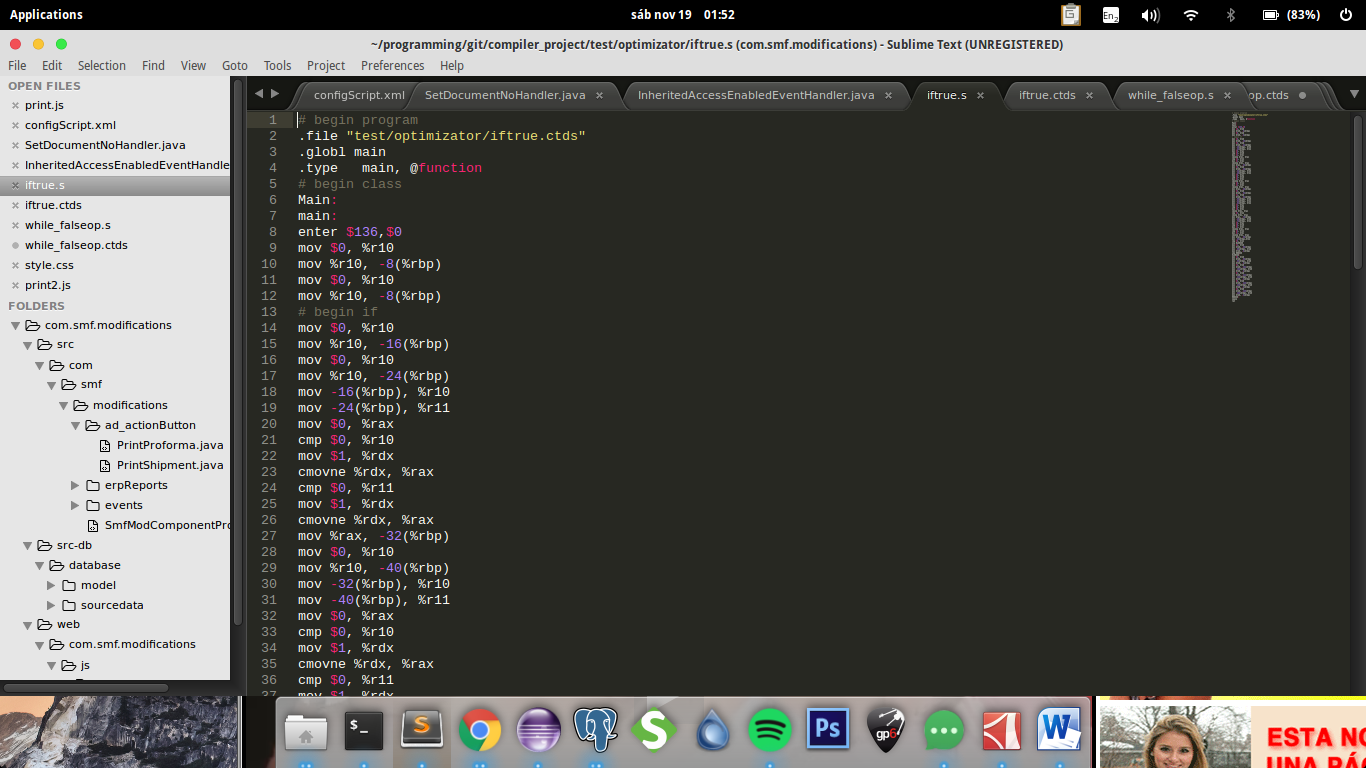
Y este es el código con optimizaciones, como se puede apreciar las líneas de código dentro del bloque del while no fueron traducidas a assembler.

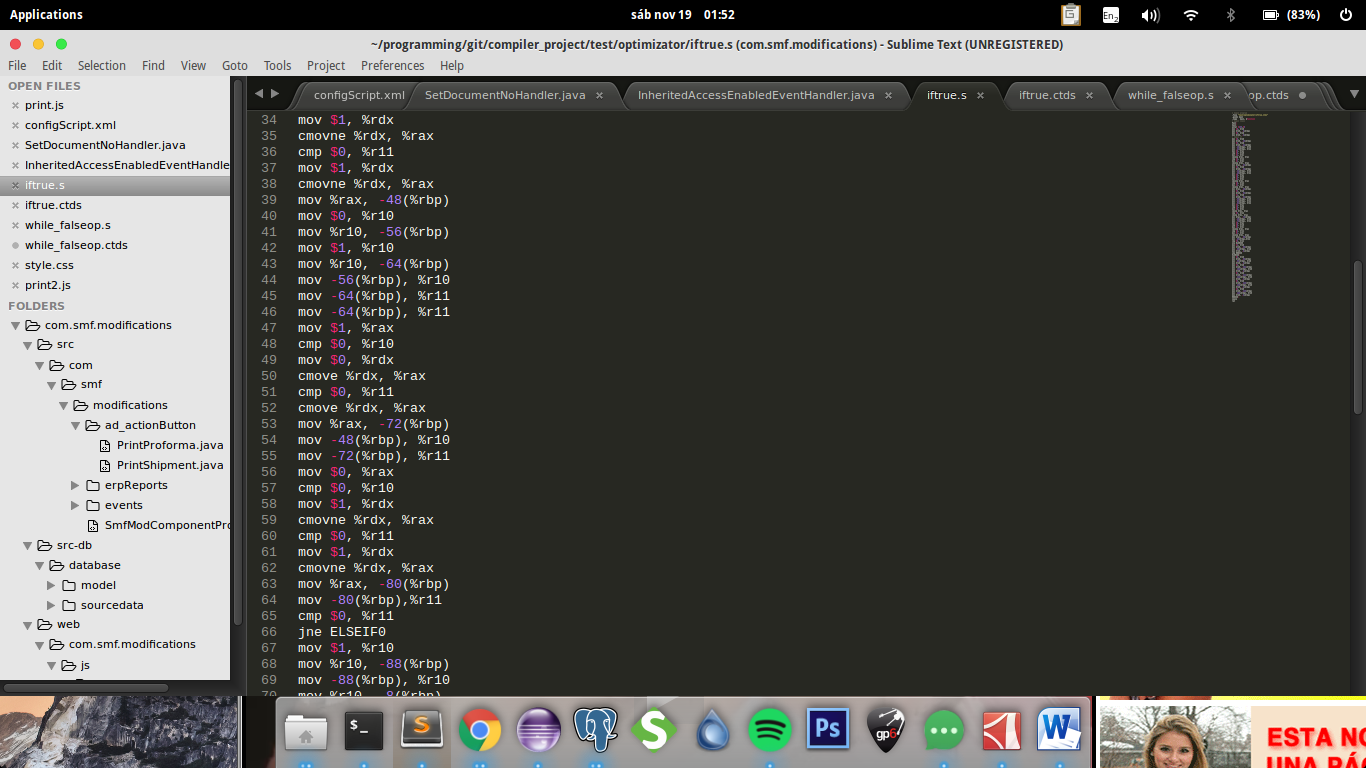


Otro ejemplo, en este caso probando el if (true) then else, dónde la condición es una expresión compleja pero conformada íntegramente por constantes, sirve para probar ambas optimizaciones.

****

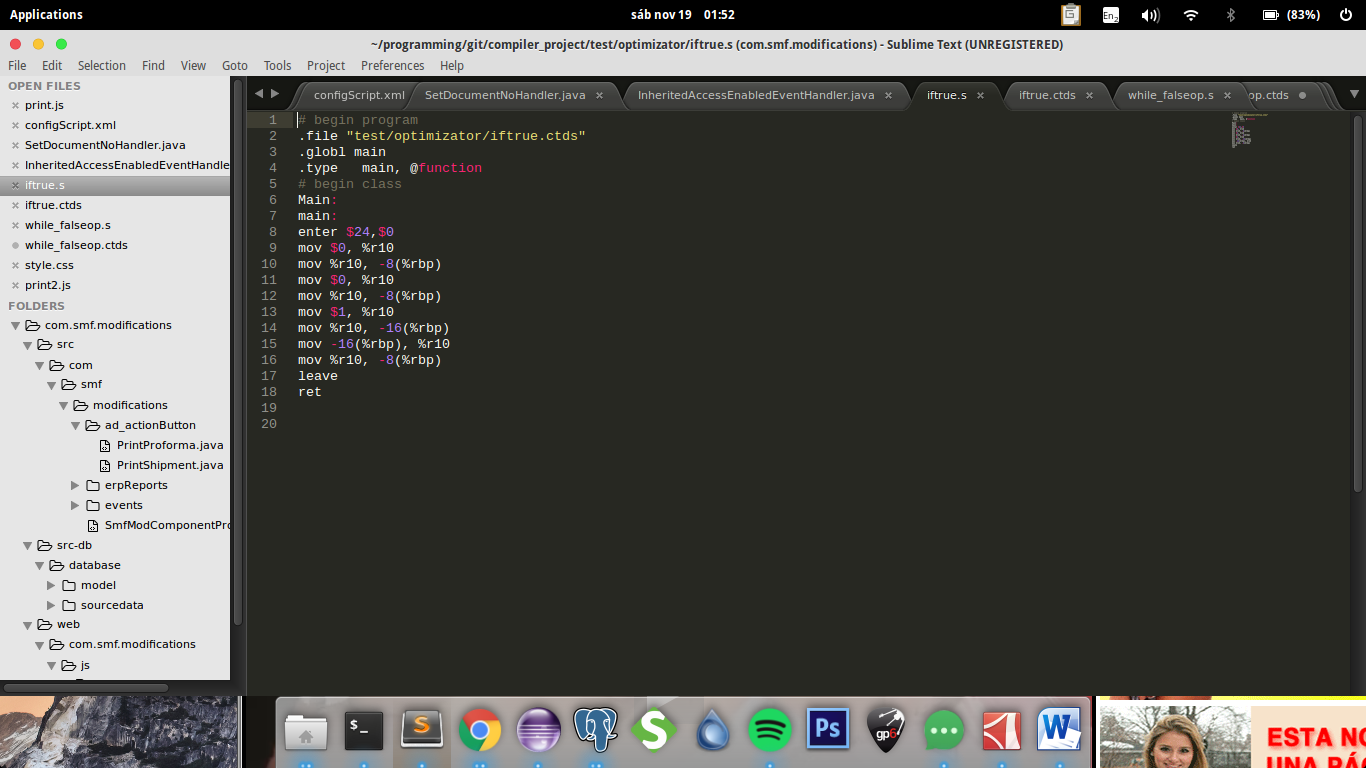
Este es el código assembler sin las optimizaciones, donde se realiza todo el cálculo para obtener el resultado de la expresión booleana y luego mediante etiquetas se definen los bloques then y else.

******

******

******

Y aquí el código generado con las optimizaciones, como se puede apreciar, no se realiza el cálculo de la expresión booleana y los labels no son necesarios ya que siempre se va a ejecutar el código del bloque then, por lo que puede obviarse toda la estructura del if y transcribir sólo las líneas contenidas en ese bloque.

******

***Detalles propios del lenguaje***

El lenguaje soporta todas las operaciones descriptas en la especificación (archive 01-TDS-spec-lenguaje.pdf).

No esta contemplado el uso de variables de un tipo no básico como parámetros en métodos, ni como atributos de clases. En el caso del uso de arreglos como atributos de una clase, sintácticamente está permitido, pero la generacion de código assembler no se ha llegado a implementar. En tanto en lo que la clase Main (clase que contiene el método main) respecta, los atributos de la misma no son implementados en la etapa de generación de código assembler.

El compilador está preparado para soportar instrucciones con variables float solo que la traducción a assembler de las mismas no fue realizada por el sólo hecho de priorizar la etapa de optimización.