Universidad Nacional de Río Cuarto

Facultad de Ciencias Exactas, Físico-Química y Naturales

Taller de Diseños de Software 2014

Proyecto: Compilador

Integrantes:

- Coria, Gastón

- Gastaldi, Diego

Índice:

Introducción:

El proyecto consiste en el diseño e implementación de un compilador para un lenguaje de programación simple, denominado C-TDS.

Dicho desarrollo se dividió en diferentes etapas, las cuales serán detalladas en este informe, en las cuales se utilizaron las siguientes herramientas:

- Jflex: permitiendo principalmente realizar un análisis léxicos aunque también tiene otras funcionalidades como la de participar en el análisis semántico.
- Cup: el cual se utilizó para realizar el análisis sintáctico y semántico principalmente.
- El lenguaje java y sus librerías para el desarrollo de estructuras y representar ciertos comportamientos del compilador como son los alcances de los identificadores, entre otras cosas.
- Git: utilizamos un repositorio de Git desde el cual realizamos las modificaciones y actualizaciones al proyecto y llevar un control de versiones desde este.
- Umbrello: para representar el esquema del AST y de los símbolos utilizados en la tabla como se describirá mas adelante en UML.
- Compilador gcc y el lenguaje C: Una vez alcanzada la etapa de generación de código assembler, para ayudarnos a comprender el comportamiento del lenguaje a crear y la traducción desde el que estamos generando.

Distribución del trabajo:

Aquí, al principio decidimos conjuntamente el diseño de esta etapa, y dividimos diferentes clases o métodos para ser implementadas por los miembros del grupo.
Fuimos implementando diferentes métodos previamente acordado y, en ocasiones, consultándonos entre nosotros y con los docentes ante el surgimiento de dudas.

Etapas:

Análisis Léxico y Sintáctico:

Decisiones:

La descripción de los comentarios multilinea en Jflex la llevamos a cabo mediante la creación de un nuevo estado en el autómata que crea jflex, permitiendo así, llegar a este nuevo estado cuando se alcanza una cadena “/*”, y luego, en este consumir todo tipo de símbolos hasta encontrar la cadena "*/", que lleva de nuevo a los estados creados por Jflex. Este acepta los comentarios multilinea al estilo java, de manera que al encontrarse la cadena "/*", consume todos los símbolos siguientes hasta encontrar un "*/", donde finaliza el mismo. Notar que "/*" dentro del comentario es ignorado.

La desambiguación de la gramática nos llevó a tomar decisiones como la de definir precedencias, y en casos más complejos, tuvimos que factorizar ciertas producciones.

Descripción de diseño:

En esta etapa desarrollamos un archivo con extensión .Flex donde escribimos todas las expresiones regulares que permiten reconocer todas las cadenas o símbolos validos que pueden aparecer en el lenguaje, un archivo con la extensión .cup donde representamos la gramáticas y eliminamos las ambigüedades de esta, aquí también agregamos código java para informe de errores. Y por último, una clase "Main" que crea los analizadores y

permite testearlos.

Problemas:

Informar línea de los errores: Lo solucionamos incorporando la línea y la columna de cada símbolo reconocido en su creación y definiendo un par de métodos en el parser para que "lea" estos y los muestre.

Ambigüedades de la gramática: Logramos eliminarla definiendo precedencias y factorizando. En el caso de las primeras, también tuvimos ciertos errores con esta ya que las habíamos utilizado con símbolos que estaban en diferentes niveles y cup, luego de consultar, vimos que deben estar todos al mismo nivel para que estas se respeten.

Definición de expresiones regulares para los comentarios: En un principio intentamos hacer una expresión regular que logre aceptar los comentarios, pero, luego de investigar un poco vimos cómo crear un nuevo estado y decidimos utilizar esto, tras no poder definir una expresión regular que haga lo deseado.

Entre los menos importantes en cuanto al desarrollo del compilador pero no menos difíciles, tuvimos problemas con la forma de definir y escribir la gramática y los símbolos, poder correr test y así probar lo desarrollado, crear un script para correr estos, entre otros.

Análisis Semántico:

Decisiones:

Durante la implementación de la tabla de símbolos, elegimos utilizar una lista de listas de símbolos dejando de lado por ejemplo la estructura hashmap, ya que si bien es un poco menos eficiente, a las listas las conocíamos más y así pudimos agilizar el trabajo.

Al desarrollar los símbolos a ingresar en la tabla mencionada anteriormente, decidimos hacer uso de la herencia de clases y así poder distinguir entre las funciones, arreglos y variables simples, teniendo en cuenta que cada uno de estos tiene diferentes atributos. Otra opción era la de crear todos los atributos y solo setear los que utiliza cada tipo pero consideramos que la primera forma mencionada era más adecuada y ventajosa.

Otra decisión fue la de darle diferentes niveles a los nombres de los métodos que a sus parámetros, es decir, los nombres de los métodos perteneces al nivel en el que se encuentra el parser actualmente, mientras que sus parámetros se encuentran en un nivel superior a este, evitando así que si hay presente una variable con el mismo nombre que alguna de los parámetros, no sea tomado como re declaración.
A su vez, las variables que se declaren en el método antes mencionados pertenecerán a un nivel aun mayor, dando la posibilidad de definir variables de igual nombre que los parámetros (esto "mataría" al parámetro allí).

En cuanto a la construcción de AST, usando como base el árbol que se nos brindó, necesitamos agregar varias clases como la de llamadas a métodos, las diferentes asignaciones, los ciclos, entre otras. Aquí en general decidimos hacer uso de la herencia y así, poder simplificar la implementación del Visitor (patrón de diseño usado para realizar chequeos de tipo semánticos).
En el caso de los literales, teníamos la opción de integrar a estos los StringLiteral a los ya existentes como son FloatLiteral, IntLiteral y BoolLiteral o diferenciarlos de estos. La elección fue la segunda, los motivos fueron que los literales “string” no pueden ser usados en todos los lugares que si pueden los demás literales, los string solo pueden ser usados en las llamadas a los métodos externos, por ello la diferenciación.

Una vez creado el árbol, decidimos guardarlo en una LinkedList, donde cada elemento es un árbol que tiene toda la información de un método determinado. Para este heredamos del símbolo que usamos para representar a las funciones y le agregamos como atributo el árbol, así obtenemos toda la información del método como son el tipo devuelto, los parámetros, el nombre, entre otros.

En cuanto a la semántica del lenguaje, tanto en las operaciones, como en las asignaciones decidimos que los operandos sean iguales, para simplicidad, a excepción de la operación “div”, que en un principio, por más que los operandos sean de tipo “int”, el tipo de la expresión debía ser de tipo “float” pero luego lo cambiamos para que se realice la división entera, quedando así todas las operaciones con sus operandos y resultados del mismo tipo.

Descripción del diseño:

Tabla de símbolos: Utilizamos la estructura LinkedList provista por la librería "útil" de java, en la cual, cada elemento de esta son listas del mismo tipo que guardan los identificadores que hay en el alcance. El primero de estos es el alcance actual. En esta clase, además implementamos diferentes métodos que permite agregar y eliminar el nivel superior, agregar un símbolo al nivel superior, buscar un símbolo desde el superior hasta el inferior hasta que lo encuentre e insertar un conjunto de símbolos de manera conjunta.

Símbolos: estos son usados para representar diferentes estructuras como son los arreglos, las funciones y los símbolos simples, que van a ser insertados en la tabla. En la implementación creamos una clase abstracta que tiene los atributos comunes de todas las estructuras, y diferentes clases concretas que agregan los atributos propios de cada clase.

AST: las llamadas a los métodos las diferenciamos teniendo en cuenta si forman parte de una expresión o si es una sentencia. Y en cada caso creamos las clases que representan a las llamadas internas y a las externas. Aquí, estas últimas heredan de la clase abstracta mencionada al principio (llamada a métodos).

AST: En el caso de los ciclos, claramente son sentencias, por lo que heredan de la clase con el mismo nombre. Aquí creamos una clase abstracta y cada ciclo aceptado por el

lenguaje hereda de esta. En principio solo son el "for" y el "while".

Un punto importante del diseño de esta etapa es la implementación del patrón de diseño Visitor. Para esto creamos las siguientes clases concretas de visitor:
- TypeCheckVisitor: Donde realizamos los chequeos de tipos de las operaciones, asignación y expresiones que participan en ciertas sentencias como por ejemplo, la condición del if.
Aquí, a grandes rasgos, cada método "visit" retorna su tipo y, a su vez, comprueban que sus componentes no tengas errores de este estilo. Además, como se requirió en etapas superiores, seteamos los tipos que tiene las expresiones para su uso posteriormente.
- ReturnCheckVisitor: Esta clase corrobora que todos los métodos pueden acceder siempre a alguna expresión "return", y que esta retorne un tipo que se corresponda con la definición del método donde se encuentra.
Aquí, cada "visit" retorna un "boolean", el cual indica, si se trata de una sentencia, que en ella hay un "return" con el tipo correcto o no. En el caso de las expresiones, indica si son del mismo tipo que el del retorno de la expresión.

Problemas:

Al intentar usar la tabla de símbolos en el archivo .cup, nos dimos cuenta que la gramática, en el intento por desambiguarla, había quedado difícil de seguir por donde avanza el "parseo" y así poder identificar las declaraciones y los alcances de estas. Por lo que la reescribimos para facilitar lo antes mencionado. Otra opción era la de utilizar varios atributos estáticos para guardar información de los identificadores como son: el nombre, el tipo, entre otras. Pero consideramos que esto iba a complicar cada vez más la legibilidad del código.

Ambigüedades con la gramática al introducir código de Java entre llaves: cuando necesitamos crear un nuevo nivel o tratar un símbolo mediante código Java, ocasionalmente, al compilar con cup, obteníamos ambigüedades en la gramática que pudimos solucionar sin modificarla nuevamente.

Otro inconveniente aquí, fue el de tener dos secciones de código java entre llaves de

manera consecutiva, aquí, como el error no nos daba la información necesaria para poder darnos cuenta de ello, se requirió mucho esfuerzo para encontrar donde estaba dicho error.

Al generar el AST en el archivo cup, tuvimos incontinentes en los tipos que devolvían cada terminal y no terminal de la gramática por lo que vimos la necesidad de cambiarlos y en ocasiones modificar el diseño del AST introduciendo nuevas clases o modificándolas.

Generación de código intermedio:

Decisiones:

Posiblemente, la decisión más importante en esta etapa fue la de volver a utilizar el patrón Visitor para generar las instrucciones. Esto nos facilitó la implementación de esta etapa ya que lo conocíamos de las etapas anteriores.

Aquí, a grandes rasgos, este toma un completeFunction y a partir de este genera una lista de “Instr”, que representan el código intermedio. Los métodos del Visitor decidimos que devuelva un entero que representa la dirección de memoria que contiene el resultado de lo calculado por dicho método. A partir de esto pudimos ver que las sentencia retornaran siempre “null” y las expresiones el resultado de lo calculado por estas.

En el caso de las instrucciones a crear, buscamos que sean lo suficientemente abstractas para no complicar demasiado esta etapa, pero que no disten en demasía de las instrucciones del código assembler para que en la próxima etapa no sea muy complicado traducirlas.

Dirección de variables locales: para tener las direcciones de memoria de las variable locales, modificamos el archivo “.cup” y los símbolos usados en la tabla, donde cada vez que encuentra una declaración de una variable local, al constructor del símbolo que va a crear le agregamos un atributo que es la dirección del mismo en la memoria, entonces, cada vez que alcanza un “location”, a partir del símbolo, le asigna la dirección a dicho “location” teniendo así para cada símbolo una única direcciones de memoria. Estas direcciones son generadas mediante la clase “Labels”. Una vez llegado al final de la definición del método, al crear el “completeFunction”, le agregamos al constructor la cantidad de memoria para las variables locales.

Calculo de memoria a reservar para las variables locales: para generar el código de una función, reinicializamos el objeto de la clase “Labels” que usamos para generar la dirección de las variables auxiliares del mismo, a partir de la cantidad de parámetros del método más las variables locales del mismo. También, antes de generar las instrucciones del método, guardamos la posición en la lista de instrucciones donde tenemos que poner

las instrucciones. Una vez hecho esto, generamos las instrucciones del método, y al finalizar esto, calculamos la cantidad de espacio total a reservar para la función teniendo en cuenta variables locales, temporales y parámetros, y generamos la cabecera del método en la posición que obtuvimos antes.

Descripción del Diseño:

Visitor: Para generar las instrucciones de código intermedio hicimos una nueva implementación del Visitor, en la cual vamos generando las instrucciones y devolviendo el Integer que representa la dirección donde se guarda el resultado de estas. A su vez, estas instrucciones las guardamos en una lista, la cual será usada cuando hagamos uso del código generado. Aquí requerimos hacer un tratamiento especial en el llamado a este, donde antes de comenzar con la generación mediante los métodos de esta clase, creamos las instrucciones de encabezado del método.

Clase Instr: Para representar las instrucciones creamos una clase con tres atributos de tipo String que representan los operando y otro de un tipo enumerado que representa la operación a realizar.

Clase Labels: En cuanto a la generación de etiquetas para la realización de ciclos por ejemplo, creamos una clase un método llamado getLabel, el cual, cada vez que es llamado, retorna un String distinto. Además esta clase será usada para generar diferentes direcciones de memoria en la etapa siguiente.

Clase Operand: Clase de tipo enumerada que contiene todas las instrucciones del código de tres direcciones.

Modificamos las clases del paquete semanticAnalizer, agregándoles un atributo que represente la dirección del símbolo, usando esto en el archivo “.cup” como se indicó en la sección decisiones.

Problemas:

Para las sentencias "break" y "continue", las cuales necesitan saber a qué etiqueta saltar,

es decir, si la sentencia es un "break", se requiere saber cómo se llama la etiqueta de fin del ciclo más interno donde esta sentencia se encuentra. El caso de la "continue", es similar, solo que se necesita la etiqueta de inicio de ciclo. Para solucionar esto, introdujimos una lista en el Visitor, en la cual, cada vez que se alcanza una sentencia "while" o "for", se le ingresa a la lista tanto la etiqueta de inicio como de finalización de ciclo, en ese orden, y si en este hay un "break" o "continue", obtienen la información que necesitan antes detallada de esta lista.

Llamadas externas y Llamadas internas: Aquí, calculamos las instrucciones para calcular los parámetros y una vez hecho esto lo apilamos dicho parámetro, pero, el problema fue que si el parámetro es otra llamada a un método, este usara los registros para su llamada y pisara los valores que ya se habían cargado para la llamada actual. Esto lo solucionamos calculando todos los parámetros primero y luego apilar los mismos.

Asignaciones y locations: En esta situación, como cada método del visitor retorna la dirección de memoria donde se guarda el resultado de la operación que se realiza, en el caso de los “locations”, retornamos la dirección del mismo, pero en el caso de los “arrayLocations”, tenemos el índice en una dirección por lo que no podemos calcular la dirección concreta que deseamos. Por esto decidimos asignar dicho valor a una dirección de memoria y retornar esta. Esto trajo como consecuencia que si le queríamos asignar algún valor al “arrayLocation”, en realidad se lo asignaría a la dirección nueva generada. Viendo esto decidimos tratar este caso particular donde necesitamos asignarle un valor a un array en las asignaciones y no llamar al visitor de estos. En cualquier otro caso donde se requiera el valor que tiene el array en algún index, si se realizara la llamada.

Calculo de espacio para variables y creación de cabecera del método: Como retrasamos la creación de la cabecera del método (label, reservación de memoria, tratamiento del stack) hasta tener calculado el espacio a reservar, si dentro del método hay una llamada externa, como estas tienen un string literal, y la definición de este se coloca al principio de la lista, entonces, la cabecera del método deberá colocarse en una posición distinta a la que se había calculado antes, más precisamente, un posición más por cada llamada a métodos externo que tenga dentro. Esto en un principio lo habíamos obviado y generaba las funciones de manera desordenada.

Generación de código assembler:

Decisiones:

Uso de 64 bits: El motivo de la utilización de 64 bits no es más que como fuimos viendo ejemplos generados a partir de programas de C y estos fueron generados con esta cantidad de bits y además, en el tutorial que seguimos tenia ejemplos con 64 bits también, por lo que decidimos que nuestro compilador genere este tipo de assembler.
Además, teniendo en cuenta que en Organización del Procesador estuvimos trabajando con assembler de 32 bits y queríamos conocer una nueva estructura.

Nueva clase paramRegister: Como se detalló en el inciso anterior, se utilizó assembler 64 bits, y en este, para las llamadas a las funciones, los parámetros se mueven en principio en ciertos registros con esta función, pero, si la cantidad de parámetros es mayor a la cantidad de estos registros, se comienza a utilizar la memoria. Para lo primero, decidimos crear una clase de tipo enumerada que contenga dichos registro de manera ordenada para poder acceder a ellos sin necesidad de manipular sus nombres, sino que solo teniendo en cuenta la posición del parámetro a apilar.

Variables Globales: La generación de código assembler para estas variables trajo varios problemas y la necesidad de tomar varias decisiones. En un principio tuvimos la necesidad de distinguir los símbolos que son globales de los que no los son, esto nos llevó a agregarle un atributo tanto a los símbolos que son almacenados en la tabla de símbolos como a los Location del árbol, el cual es de tipo Boolean. Esta diferenciación fue necesaria ya que las operaciones que contengan variables globales ser llevaran a cabo de diferente manera que las que tengan variables locales. Para definir estas variables que se están describiendo, requerimos almacenar estas en una lista para luego recorriéndola, ir generando el código que las definen en assembler. Aquí, la lista se generó en “.cup” y se recorrió en la clase InstrCodeGenVisitor, generando las instrucciones para poder definirlas.
Al requerir un tratamiento distinto a las variables locales, para los métodos de la clase InstCodeGenVisitor, cada vez que se alcanza a un Location, se debe distinguir si este es global o no y a partir de esto generar la instrucción correspondiente.

Descripción del diseño:

Clase genAssemblyCode: Esta clase, a partir de una lista de instrucciones generadas en la etapa anterior, la recorre y genera el código asembler correspondiente teniendo en cuenta la finalidad de cada una previamente acordada. Todas las instrucciones generadas son almacenadas en una variable global que luego será retornada.

Modificaciones en InstCodeGenVisitor:
- Agregamos métodos para la inicialización de las variables tanto globales como locales con los valores por defectos descriptos en la consigna del proyecto. Esto lo realizamos moviendo a cada dirección de las variables el valor por defecto que tiene de acuerdo con el tipo de esta al inicio del método donde fue declara. En el caso de las variables globales, son inicializadas en el método Main.
Para poder llevar a cabo esta tarea, se requirió, además, crear nuevas instrucciones del código intermedio.
- Incorporación de las variables globales: acá también requerimos la implementaron nuevas instrucciones de código intermedio para llevar a cabo esto, además se necesitó modificar los métodos de los “Location” para tener en cuenta este tipo de variables y en algunos casos, también los métodos que requieren la direcciones de estas variables.

Clase paramRegister: Como se mencionó en la sección de decisiones de esta etapa, esta clase contiene los registros que se utilizan para el pasaje de parámetro permitiendo facilitar las llamadas a funciones.

Modificaciones en el archivo “.cup”: Principalmente estas se deben a la incorporación de la distinción de las variables globales con las locales. Esto fue necesario ya que en assembler recién un tratamiento diferente.

Instrucciones “and” y “or”: En el caso de estas instrucciones tuvimos que decir entre crear muchas instrucciones en código de tres direcciones o, aprovechar la abstracción de estas y pasar al generador de código asembler mucha información en comparación con las otras instrucciones. En ambos casos consideramos que se “ensusiaría” el código y preferimos darle prioridad a la abstracción del código de tres direcciones y dejar el trabajo sucio al generador del asembler ya que en caso de elegir la otra opción el código de tres direcciones no sería lo suficientemente abstracto y al crear más instrucciones también

deberíamos usarlas en las otras operaciones y sentencias, perdiendo así su abstracción, y por consiguiente, sus beneficios.

Problemas:

Parámetros de llamadas a métodos: En un principio apilábamos los parámetros solo en la memoria, lo que nos trajo problemas con las llamadas externas, principalmente con método “printf”, ya que buscaba los parámetros en los registros destinados a almacenar los mismos. Por lo que decidimos cambiar este comportamiento para todas las llamadas, tanto internas como externas, y así poder tratar a ambas de la misma manera. Esto lo logramos mediante la implementación de la clase paramRegister que se describió anteriormente.

Lenguaje assembler (registros y operaciones): Los problemas de esta etapa principalmente se centraron en el lenguaje assembler, y en encontrar los errores que se generaban con este lenguaje. Entre ellos podemos nombrar violaciones de segmentos por errores de reserva de memoria, accediendo a los índices de los arreglos con registros incorrectos, pasando mal los parámetros para las llamadas externas, y otros errores generados por el mal uso de las variables globales o la mala definición de las mismas.

Diferentes etiquetas: En un principio, generábamos todas las etiquetas de la misma manera hasta que pudimos observar mediante ejemplos que realizamos a partir del lenguaje C que las etiquetas de los métodos tiene una pequeña diferencia con las otras (el punto inicial).

Tipo de los operandos de las instrucciones: La definición inicial de los operandos de las instrucciones del código de tres direcciones eran de tipo String, ya que la idea era que se pase la dirección tanto del resultado como de los operandos, pero al avanzar con la implementación, pudimos observar que requeríamos mayor abstracción para esto, como es el caso de las operaciones and y or que se mencionó en esta etapa. Así, solucionamos el problema cambiando el tipo de dichos atributos a Object y así poder usar luego el tipo más conveniente para cada instrucción.

Inicialización de variables: Como indica la descripción del proyecto, los valores enteros deben ser inicializados con el valor cero y los booleanos con el valor “false” (el cual lo

representamos con un cero también). Para ello tuvimos que diferenciar las direcciones de memoria utilizadas para variables locales con el resto y a estas asignarles cero. El problema estuvo en la primer parte, es decir, diferenciar las direcciones de memoria reservadas y decidir donde llevar a cabo dicha inicialización. Para solucionarlos, al empezar a generar el código asembler para el método Main (el cual sabemos que todas las clases lo tienen), implementamos un método para que lleve a cabo dicha inicialización a las direcciones de memoria que pudimos distinguir ya que la reserva se realiza empezando por los parámetros, luego las variables locales y al final los temporales.

Generación de assembler para float:

Decisiones:

Lugar de diferenciación de las variables de tipo float: teníamos la opción de diferenciar al generar el código intermedio o al generar el código assembler. Optamos por la primera, creando más instrucciones en el código intermedio, replicando las operaciones usadas para los enteros pero que luego, en el assembler utilizaran las instrucciones y los registros destinados para las variables de este tipo. Para ellos, en los métodos del visitor que lleva a cabo esta tarea, en cada acceso a un “Location” o a un “Literal” necesitamos corroborar de qué tipos son y a partir de esto, seleccionamos la opción correspondiente. Esta elección se debió a que en una primera impresión, no había mucha diferencia entre la implementación de las opciones, pero vimos que si elegíamos la otra opción, debíamos tener muchos casos al crear las instrucciones de assembler y como consecuencia de ello, se iba a “ensuciar el código” de la clase que genera assembler, siendo esta la clase critica en esta etapa y consideramos necesario tener una mejor legibilidad aquí. En el caso de la opción que elegimos, si bien tenemos más instrucciones de código intermedio, la diferenciación entre los tipos acá la íbamos a tener que hacer igual teniendo que crear las variables float de manera diferente.

Nuevas operaciones en el código intermedio: como se detalló en el punto anterior, se requirió la creación de instrucciones para el tratamiento de variables “float”, para la diferenciación entre las llamadas externas e internas, entre otros.

Descripción del diseño:

InstCodeGenVisitor: Como se describió en la sección decisiones de esta etapa, en esta clase se realizaron los casos de los tipos de las variables o constantes y de acuerdo a estos, se decidieron que instrucciones usar para ellos.

genAssemblyCode: Agregamos todas las operaciones necesarias para tratar las variables float a partir de ejemplos generados desde el lenguaje C, al igual de como lo hicimos con el tipo Int en etapas anteriores.

paramRegister: Agregamos un nuevo arreglo con los registros destinados al pasaje de parámetro para las variables reales de manera ordenada para llevar a cabo la llamada a los métodos con parámetros de este tipo.

Problemas:

Parámetros tipos float: al mostrar una variable float con el método printf mostraba 0.0 siempre, sin importa el valor que tenga esta. Esto lo solucionamos agregándole una instrucción antes del llamado al método printf, donde se le indica la cantidad de parámetro que tiene que buscar en los registros xmm, es decir, si a rax le guardamos un 1, entonces el método buscara en xmm0 el parámetro y no en rdi (donde guarda los enteros).

Reservación de memoria múltiplo de 16: para hacer uso del método printf, en ocasiones nos daba violación de segmento, lo cual era debido a que la memoria a reservar en el método debe ser múltiplo de 16 y no lo teníamos en cuanta. Esto lo solucionamos comprobando esto al momento de reservar la memoria.

Más parámetros que la cantidad de registros que hay: al momento de apilar los registros, lo hacíamos de manera incorrecta, por lo que al ir a buscarlos, los recuperaba de manera errónea. Esto se debía que al apilarlos no utilizábamos el registro correcto (usábamos el rbp) y no el rsp como debería ser. Además de esto, lo apilábamos en orden inverso, lo cual se solucionó cambiando el modo en el que se recorrían los parámetros.

Lugar de incremento del índice de un ciclo for (problemas al realizar un continue: ciclo infinito): Al alcanzar una instrucción “continue”, la ejecución volvía al label de inicio del ciclo y no llegaba a las instrucciones que hacían incrementar el índice del ciclo.

Direcciones de los locations: cuando se tenía que hacer una asignación a un location, debido a la manera en que está diseñado el visitor de la clase InstCodeGenVisitor, obteníamos una dirección que tenía el contenido del location pero no era la dirección de este, sino una copia del contenido, por lo que asignarlo en esta dirección no modificaba el contenido del location. Para solucionar este problema, “hardcodeamos” el código en los tres tipos de asignaciones, donde, en vez de llamar el método accept de los locations, los tratamos ahí mismo sin usar su visitor.

Optimización:

Las optimizaciones son ejecutadas por una nueva clase para poder observar las diferencias de los códigos generados con y sin optimizaciones. A excepción de la optimización “Reutilización de memoria”, la cual, como requirió la modificación de dos clases y no la creación de nuevas, forma parte de ambas clases “Main”. Por esto, para genera los ejemplos, comentamos parte del código.

Cálculos constantes:

Esta optimización consiste en realizar los cálculos constantes antes de generar el código asembler, así, evitamos generar instrucciones para calcular algo que lo podemos tener a priori.
Esto lo llevamos a cabo mediante la implementación de otra clase, en la cual implementamos nuevamente el patrón visitor para recorrer el árbol sintáctico y alcanzar operaciones entre valores constantes u operaciones donde se puede deducir el resultado (por ejemplo un “and” con uno de los operandos falsos).

Código muerto:

Esta optimización consiste en eliminar las instrucciones que nunca van a formar parte del hilo de ejecución de un programa, es decir, nunca se van a ejecutar, como puede ser el caso de un “if (true)”, donde el bloque “else” nunca se va a ejecutar y mediante esta optimización, es borrado.
La implementación de este, la realizamos, al igual que la anterior, mediante la utilización de un visitor.

Reutilización de memoria:

En algunas operaciones, es necesario usar variables temporales, las cuales solo son usados para guardar un determinado valor para hacer una operación y no son vueltas a usar. Por ello, buscamos reutilizar estas direcciones y así, que el programa requiera menos memoria para su ejecución.

Ejemplo:

Para el test:

```
class testIf {
  void if1() {
    int a;
    if (4 - 3 * 2 > 0 * 4 -10) {
       externinvk("printf", int, "Parte if");
    } else {
       externinvk("printf", int, "Parte else");
    }
    return ;
    a = a + 2;
}
void main() {
  if1();
  return;
}
}
```

Código generado sin optimización:

```
.SL3:
                    .string "Parte else"
.SL1:
                    .string "Parte if"
.text
.globl    if1
.type     if1, @function
if1:
enter               $(8 * 14), $0
movq                $0, %r10
mov                 %r10, -8(%rbp)
movq                $4, -24(%rbp)
movq                $3, -32(%rbp)
movq                $2, -40(%rbp)
mov                 -32(%rbp), %r10
mov                 -40(%rbp), %r11
imul                %r11, %r10
mov                 %r10, -48(%rbp)
mov                 -48(%rbp), %r10
mov                 -24(%rbp), %r11
sub                 %r10, %r11
```

```
mov		%r11, -56(%rbp)
movq		$0, -64(%rbp)
movq		$4, -72(%rbp)
mov		-64(%rbp), %r10
mov		-72(%rbp), %r11
imul		%r11, %r10
mov		%r10, -80(%rbp)
movq		$10, -88(%rbp)
mov		-88(%rbp), %r10
mov		-80(%rbp), %r11
sub		%r10, %r11
mov		%r11, -96(%rbp)
mov		-56(%rbp), %rax
cmp		-96(%rbp), %rax
setg		%al
movzb		%al, %rax
mov		%rax, -104(%rbp)
movq		$1, -112(%rbp)
mov		-112(%rbp), %r10
cmp		-104(%rbp), %r10
jne		.falseCondL0
mov		$.SL1, %r10
mov		%r10, %rdi
mov		$0, %rax
call		printf
jmp		.endIfL2
.falseCondL0:
mov		$.SL3, %r10
mov		%r10, %rdi
mov		$0, %rax
call		printf
.endIfL2:
mov		$0, %rax
leave
ret
movq		$2, -112(%rbp)
mov		-16(%rbp), %r10
mov		-112(%rbp), %r11
add		%r10, %r11
mov		%r11, -112(%rbp)
mov		-112(%rbp), %r10
mov		%r10, -16(%rbp)
.globl	main
.type	main, @function
main:
enter		$(8 * 2), $0
mov		$0, %rax
```

```
call            if1
mov             $0, %rax
leave
ret
```

Código generado con las optimizaciones:

```
.SL0:
                .string "Parte if"
.text
.globl  if1
.type   if1, @function
if1:
enter           $(8 * 2), $0
movq            $0, %r10
mov             %r10, -8(%rbp)
mov             $.SL0, %r10
mov             %r10, %rdi
mov             $0, %rax
call            printf
mov             $0, %rax
leave
ret
.globl  main
.type   main, @function
main:
enter   $(8 * 2), $0
mov             $0, %rax
call            if1
mov             $0, %rax
leave
ret
```