Universidad Nacional de Río Cuarto
Facultad de Ciencias Exactas, Físico-Química y Naturales
Departamento de Computación
Taller de Diseño de Software
(Cód. 3306)

Compilador C-TDS

Arangue Ezequiel
Cibils Juan Ignacio

2016

1 Introducción

El proyecto de la materia consiste en la implementación de un compilador para un lenguaje orientado a objetos simple, similar a C, llamado C-TDS.

Las especificaciones del lenguaje respecto a Léxico, Gramática y Semántica se encuentran reflejadas en el documento de especificaciones (*01-TDS-spec-leng.pdf*) provisto por la cátedra, cuyo contenido guió el desarrollo de las diferentes etapas del proyecto.

2 Etapas del Proyecto

2.1 Análisis Léxico - Sintáctico

El objetivo de esta etapa es la construcción del analizador léxico del lenguaje o Scanner, el cual a partir del código fuente de un programa C-TDS debe reconocer los símbolos del lenguaje y retornar *tokens.* Un *token* representa a una clase de símbolos del lenguaje. Los símbolos que no son *tokens* son descartados. La herramienta utilizada para generación del Scanner es JFlex.

A partir de los tokens generados por el analizador léxico, realizamos el Analizador Sintáctico o Parser. El Parser toma como entrada la secuencia de tokens y verifica que esta secuencia sea una secuencia válida, es decir, que cumpla con la especificación sintáctica del lenguaje.
La herramienta utilizada para la generación del parser es CUP. A partir de la especificación de una gramática, CUP genera el parser para el lenguaje C-TDS.
La salida de esta etapa, para un programa correcto, es un árbol sintáctico.

Decisiones de diseño e inconvenientes

- No se presentaron mayores inconvenientes en el desarrollo de esta etapa. Sólo se necesitó hacer algunas pequeñas modificaciones en etapas posteriores para adecuar el Scanner y el Parser según fuera necesario.

2.2 Análisis Semántico

Esta etapa verifica las reglas semánticas del lenguaje, por ejemplo, compatibilidad de tipos, visibilidad y alcance de los identificadores, etc. En esta etapa es necesario implementar una tabla de símbolos para mantener la información de los símbolos (identificadores) de un programa. El análisis semántico se realizará sobre el árbol sintáctico abstracto(AST) generado durante el parsing, utilizando la información almacenada en la tabla de símbolos.

La estructura del Árbol Sintáctico Abstracto (AST) es la misma que en el diagrama provisto por la cátedra y fue construido a partir del código dado en la materia.

Para el chequeo de las diversas reglas semánticas se construyeron *visitors* del Árbol Sintáctico Abstracto:

2.2.1 Main checking

El *visitor* **CheckMainExistVisitor** se encarga de controlar la cantidad de métodos **main** que posee el programa. Se espera que el programa contenga sólo un método denominado main.

2.2.2 Declarations

El *visitor* **DeclarationCheckVisitor** es quien construye la tabla de símbolos que contiene la información de los identificadores de un programa. La tabla de símbolos esta construida en forma de pila, donde se va agregando un nivel a medida que se profundiza en el recorrido del AST.
Además de la construcción de la tabla de símbolos, el *visitor* también se encarga de detectar si hay identificadores duplicados, si fueron declarados antes de ser usados, si las sentencias **continue** y **break** se encuentran en un ciclo, entre otros.

2.2.3 Type checking

El *visitor* **TypeCheckVisitor** es quien controla si los tipos se han definido de acuerdo a las reglas semánticas del lenguaje. En caso de detectar que un programa dado no cumple con alguna de las restricciones se genera un mensaje con la descripción del error.

Por lo tanto, este *visitor* se encarga de:

- controlar que el tipo de la expresión de retorno de un método coincide con el tipo de retorno.
- controlar si la condición de un **if** o **while** es de tipo **bool**.
- controlar si las expresiones de un **for**, tanto inicialización como finales sean de tipo **integer**.
- controlar que los métodos definidos como **void** no tengan una expresión de retorno.
- controlar si los operadores en las expresiones son compatibles con los tipos.
- otros.

2.2.4 Aclaraciones y limitaciones

- De acuerdo a la especificación de C-TDS, un programa es un conjunto de clases. Nuestro compilador permite la definición de un programa de esta manera pero no es posible la declaración de objetos y por lo tanto el uso de métodos de otras clases.
- No es posible tener clases vacías, estas deben contener al menos una declaración.
- Hasta el momento, no se controla el número de parámetros con el que se invoca a un método.
- Es posible definir métodos que retornen **bool** o **float** pero las expresiones de retorno no pueden contener operaciones.

  Ejemplo:

```
float name(){
  float f;
  f = 0.5;
  return f+f; -> Esta expresión arrojará error.
}
```

2.3 Generación de Código Intermedio

Esta etapa del compilador retorna una representación intermedia(IR) de bajo nivel del código. Se utilizó como código intermedio el código de tres direcciones donde cada instrucción se compone de tres elementos: los primeros dos son operandos y el tercero el resultado.
Para la generación de código intermedio se implementó un nuevo *visitor* **ICGeneratorVisitor** que recorre el AST y genera una lista de instrucciones. Luego, a partir de esta representación intermedia se generará el código objeto.

2.3.1 Inconvenientes

La etapa en sí no representó problema alguna, los incovenientes que se presentaron fueron con respecto a la etapa de Análisis Semántico ya que no se encontraba aún finalizada y por lo tanto hubo que ir desarrollando ambas en simultáneo.

2.4 Generación de Assembler

En esta etapa el compilador construye un archivo llamado “assembler.s” donde este archivo contiene el programa antes pasado por el compilador pero codificado en lenguaje assembler para luego poder compilarlo(linkearlo) con “GCC” (el compilador de “C”) y asi poder ejecutar dicho programa.

Para la generación de código assembler se implementó una nueva clase java llamada “assemblerGenerator.java” donde recibe la lista generada por la etapa anterior (lista de instrucciones de 3 direcciones) y para cada una de las instrucciones recibidas genera el código assembler según corresponda con dicha instrucción, y asi finaliza logrando un archivo con todo el codigo assembler correspondiente al programa dado dejándolo listo para luego ejecutarlo.

2.4.1 Inconvenientes

La etapa en sí no representó problema alguno, los incovenientes que se presentaron fueron falta de tiempo para poder implementar todo, al atrasarnos en la etapa de los checkeos semánticos se generaron inconvenientes de tiempo tanto para la etapa del código intermedio y en la etapa de la generación de assembler y por estos inconvenientes hubo que limitar el proyecto para poder finalizarlo.

2.4.2 Aclaraciones y limitaciones

Algunas de las limitaciones del proyecto son:

- Falto implementar todo con respecto a los Flotantes, tanto declaraciones como cualquier interacción con flotantes como por ejemplo operaciones.
- Falto implementar las llamadas a métodos.
- Entre otras. Lo que se implemento fue en cuestión de tipos lo que son enteros y booleanos, las asignaciones y declaraciones de dichos tipos, las operaciones tanto aritméticas (Ej: suma) como relacionales (Ej: mayor igual) y condicionales (Ej: and – or), los retornos, tanto el retornar un booleano o un entero o void. Los saltos fueron implementados como así también los “IF” y los ciclos (while – for) y después operaciones unarias como el incrementar, el decrementar, la negación y los labels.