Informe de
Complementos de Compilación
COOL-Compiler

Autores:

1. Yenli Gil Machado C412 y.gil@estudiantes.matcom.uh.cu
2. Liset Silva Oropesa C411 l.silva@estudiantes.matcom.uh.cu
3. Pablo A. de Armas Suárez C411 p.armas@estudiantes.matcom.uh.cu

Respositorio en Github: https://github.com/Liset97/cool-compiler-2020

Introducción:

Un **compilador** es un tipo de programa que recibe como entrada un programa en texto fuente y produce un programa en código ejecutable. Usualmente el objetivo es traducir el primer programa a código de máquina, aunque también puede ser traducido a un código intermedio o a texto. La entrada es un programa en un lenguaje que llamaremos “de alto nivel”, y la salida en un lenguaje de “bajo nivel”, que es equivalente al primero. Exactamente qué es alto y bajo nivel dependerá de muchos factores, y no existe una definición formal. De forma general, un lenguaje de alto nivel es aquel que nos es cómodo a los programadores para expresar las operaciones que nos interesa ejecutar. Así mismo, un lenguaje de bajo nivel es aquel que un dispositivo de cómputo puede ejecutar de forma eficiente.

COOL (Classroom Object Oriented Language):

Es un lenguaje de programación de computadoras diseñado por *Alexander Aiken* para su uso en un proyecto de curso de compilación de pregrado. Aunque es lo suficientemente pequeño para un proyecto de un plazo, Cool todavía tiene muchas de las características de los lenguajes de programación modernos, incluidos los objetos, la administración automática de memoria, la escritura estática fuerte y la reflexión simple . El compilador Cool de referencia está escrito en C++ , construido completamente sobre las herramientas de dominio público. Genera código para un simulador MIPS , SPIM . Por lo tanto, el idioma debería trasladarse fácilmente a otras plataformas. Se ha utilizado para enseñar compiladores en muchas instituciones y el software es estable.
Precisamente el objetivo de este proyecto es elaborar un compilador de *COOL* en $Python$. Para más información sobre el lenguaje revisar el manual de COOL en la carpeta doc.

¿Cómo Ejecutar el Proyecto? :

Para ejecutar el proyecto se realiza la siguiente instrucción:

```
$ cd source
$ ./coolc.sh <input_file.cl>
```

Donde recibe un archivo $.cl$ es decir, en **COOL**, después de que el compilador lo procese si tiene errores los mostrará en la consola y sino generará en el directorio del fichero de entrada otro fichero con el mismo nombre pero con extensión $.mips$ que contiene el código en ensamblador.

¿Qué se necesita instalar?:

- Se necesita tener instalado el $ANTLR4$.
- Además de tener $Python$

Arquitectura:

Para crear al compilador se desarrollaron los siguientes módulos:

1. $COOLLexer.py$: en este se implementa todo lo referido al análisis Lexicografico.
2. $COOLParser.py$: Aquí se procede a realizar la acción de parsing con la secuencia de tokens recibida.
3. $Visitors.py$: En este módulo se encuentra implementado tanto el análisis semántico como la generación de código.

Implementación:

El desarrollo de la implementación de nuestro compilador para COOL contó con las siguientes fases:

- $Análisis\ Lexicográfico$: Cada símbolo o palabra clave del lenguaje COOL es un $token$. En esta etapa se convierte en $tokens$ el programa de entrada y se verifica que estos pertenezcan a al conjunto de palabras claves del mismo, mediante un $Lexer$. La implementación de esta fase se encuentra en $COOLLexer.py$.

- $Análisis\ Sintáctico$: Aquí se análiza la secuencia de tokens obtenida del lexer y se produce un árbol de derivación, que nos permite determinar si esa cadena de tokens pertenece o no a nuestra gramática. El parsing que se utilizó para nuestro compilador fue el $ANTLR4$ que además utiliza algoritmos $LL(*)$ de parsing.

- $Análisis\ Semántico$: En este proceso se verifica el cumplimiento de las reglas semánticas del lenguaje y la inferencia de tipos. Para su implementación se utilizó el $Patron\ Visitor$, realizando las siguientes acciones:
    1. **TypeCOOLVisitor** se utiliza para realizar un chequeo de tipos en el árbol; es decir, agrupar los tipos de cada uno de los nodos que contiene el árbol y agregarlos a una tabla donde se forma la jerarquía de tipos.

2. **SemanticCOOLVisitor** se encarga de realizar el análisis semántico después de tener la jerarquía de tipos.

- *Generación de Código*: En esta fase, ya se tiene que el programa recibido es sintácticamente correcto y se convierte este a una serie de instrucciones a ser interpretadas por una máquina. Es el proceso de pasar de **COOL** a MIPS. En nuestro compilador pasamos directo de **COOL** a MIPS mediante la clase **CodegenVisitor** que recibe el árbol y la jerarquía de tipos para traducir las instrucciones semánticas a código de máquina.