Compilador de Cool

Daniel de la Osa Fernández — C411
Jose Luis Alvarez de la C. — C412

26 de junio de 2019

1. Compiladores

A grandes razgos, un compilador no es más que un programa, cuya entrada y salida resultan ser también programas. 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. Este proyecto tendrá como entrada programas escritos en COOL y tendrá como salida su equivalente en MIPS. De cierta manera un compilador es entonce un traductor.

2. Lenguaje COOL

El lenguaje COOL, acrónimo de **C**lassroom **O**bject-**O**riented **L**anguage; es un pequeño lenguaje de programación orientado a objetos. Este lenguaje está diseñado para poder realizarle un compilador en el transcurso de un semestre con un esfuerzo considerable. Fue confeccionado por **Alex Aiken** [1] en la **Universidad de Stanford**.

COOL tipado estático, herencia simple; es *type safe*, es un lenguaje de expresiones donde cada una posee tipo y valor. Un programa contituye un conjunto de Clases. Una clase está formada por atributos y métodos. Los atributos pueden o no estar inicializados con una expresión de tipo *asign*; pero los métodos siempre tienen una expresión *cuerpo*.

Finalmente los programas tienen la extensión *.cl*.

3. Coolc

Coolc es un compilador de **COOL** implementado en *C♯*. En este se ha utilizado **ANTLR** para generar el *lexer*, el *parser*, y las bases de las clases que implementan el patrón *visitor*. Genera código **Dot** para la visuallización del árbol de derivaciones, chequea semánticamente cada clase y; si es correcto hasta ese punto, por último genera código **MIPS**.

3.1. Preprocesamiento

Lo primero a la hora de implementar un compilador es pasar verificar si la entrada es al menos lexicográfica y sintácticamente correcta. Además se crea una estructura conocida como *Árbol de sintáxis abstracta* (AST por sus siglas en inglés). Esta será de gran utilidad facilitando las posteriores etapas de chequeo semántico y generación de código.

3.1.1. ANTLR

ANTLR es un poderoso generador de *parsers*. Es usado por grandes de la industria como **Twitter**, **Oracle** y **NetBeans**. A partir de una descripcion formal de un lenguaje a la que llamamos *gramática*, **ANTLR** genera un *lexer* y un *parser*. Entre ambos pueden reconocer cadenas pertenecientes a el lenguaje previamente definido y contruir un *parse tree* (o árbol de derivación)automáticamente. Además también genera *tree walkers* siguiendo los patrones *visitor* y *listener*. Al utilizar estos *walkers* se pueden visitar los nodos del *parse tree* y ejecutar código específico de la aplicación. En nuestro caso esta aplicación será el compilador; así que el código a ejecutar será el correspondiente a las siguientes fases del pipeline; o sea, análisis semántico y generación de código.

> **ANTLR v4** is exactly what I want in a parser generator, so I can finally get back to the problem I was originally trying to solve in the 1980s.
> Now, if I could just remember what that was.
>
> Terrence Parr. Creator of ANTLR.

3.1.2. Análisis Lexicográfico

La primera parte de un compilador es el análisis lexicográfico o *lexer*. Este lee la cadena de caracteres y los agrupa en secuencias conocidas como *lexemas*.

Para cada *lexema* se produce como salida un token de la forma $(token-name, attribute-value)$ que se le pasa a la siguiente fase (análisis sintáctico). En el *token*, el primer componente *token-name* tiene es un símbolo abstrato usado durante el análisis sintáctico; y el segundo, *attribute-value*, apunta a un elemento en la tabla de símbolos para este *token*. La información relativa a la tabla de símbolos es necesaria para el chequeo semántico y para la generación de código.

3.1.3. Análisis sintáctico

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing, as we shall see. Thus, the parser and the rest of the front end could well be implemented by a single module.

3.1.4. Gramática

En el caso de la gramática se intento que fuera lo más parecido a la definición de [1]. Salvo unas pocas excepciones como es el case de la produccion *newvar*, esto se logró. Nótese que solo se pudo alcanzar tal nivel de similitud a partir de la versión 4 de **ANTLR**. Esto es gracias a que en esta versión se resuelven gran cantidad de ambiguedades como es el caso de la *recursión izquierda inmediata* [3].

3.1.5. AST

El **AST** fue contruído sin ninguna complicación utilizando el patrón *visitor*. Se respetaron los nodos definidos en la gramática y se inicializaron los campos y propiedades necesarios en cada caso. Ejemplo de estos campos son los tipos de alguna expresión cuando era explícito o el valor de las constantes.

3.2. Análisis semántico

Para el análisis semántico se aprovechó la estructura arbólea del **AST** para definir la correctitud de cada nodo en correspondencia con la correctitud de los hijos y la propia. O sea, en cada nodo se verifica primero la correctitud de los hijos, momento en el cual se terminan de actualizar valores necesarios como puede ser el tipo de una expresión; y entonces se pasa a verificar la correctitud propia. Luego solo es necesario llamar al método **ChechSemantics()** del nodo raíz y este se encargará de todo.

3.3. Generación de Código

La generación de código es el momento en que se traducen las instrucciones a código de más bajo nivel. En nuestro caso ese lenguaje es **MIPS**. En **coolc** la generación de código se genera igualmente en forma arbólea gracias a las facilidades que otorga el **AST**. Cada nodo genera su código **MIPS** y actualiza las constantes del programa que deben de ser situadas al principio del archivo **.s**. Esto son los nombres de las clases, los *strings*, la variable **self**, etc. Cada clase posee un método constructor del mismo nombre que la clase; y cada método tiene una etiqueta asociada de la forma *clase.método:*.

3.3.1. MIPS

MIPS (siglas de **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) es el nombre de toda una familia de microprocesadores de arquitectura **RISC**. Es también la arquitectura a la que está orientada **coolc**.

3.3.2. Mars

Para probar el código generado se utilizó **MARS**, un emulador de **MIPS** con interfaz de consola. Esta interfaz fue lo que permitió la eficiencia alcanzada en el testeo del código ensamblador.

4. Conclusiones

En el presente trabajo se loograron alcanzar los objetivos del curso. Los estudiantes tuvieron que enfrentarse a un lenguaje real, con una cantidad y calidad de características adecuada para garantizar el dominio de los conocimientos adquirdos en el curso anterior de Compilación.

Referencias

[1] Alex Aiken. *The Cool Reference Manual.*

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compilers. Principles, Techniques & Tools.* (2nd edition), 2007. Addison-Wesley Professional.

[3] Terence Parr. *The Definitive ANTLR 4 Reference.* 2013. The Pragmatic Bookshelf.