Complementos de Compilación.

Amalia Ibarra Rodríguez, Gabriela Martínez Giraldo, and Sandra Martos Llanes

Universidad de La Habana,
La Habana, Cuba
Grupo C-412
{amalia.ibarra,gabriela.martinez,
sandra.martos}@estudiantes.matcom.uh.cu

Preámbulo

COOL es un lenguaje de programación orientado a objetos que, aunque pequeño, tiene muchas características relevantes de lenguajes modernos como son los objetos, el tipado dinámico y otros. El proyecto que ahora nos ocupa requiere de la implementación de un compilador completamente funcional de dicho lenguaje. El compilador debe compilar hacia MIPS y los programas resultantes deben ser ejecutados en el simulador SPIM en su versión 8.

Para nuestra implementación consideraremos varias fases: tokenización, lexer, parser, chequeo semántico y generación de código. Cada una de estas fases o módulos recibe un AST y da otro como salida. A continuación se explica de manera resumida cada una de estas y se propone una fecha para su culminación.

Plan de trabajo:

Gramática

Este primer acercamiento al lenguaje Cool requirió, como cualquier otro lenguaje, la definición de una gramática. En el archivo cool_grammar.py puede observarse como se consideró la misma para no alejarnos de lo que refería el manual.

Como ahí se puede observar, un programa de Cool consiste en una serie de definiciones de clases. Cada clase a su vez posee un conjunto de atributos y de funciones. Las expresiones que pueden formar parte de dichas funciones son el corazón del lenguaje.

No es nuestra intención hacer la historia muy larga, pues para eso está el manual, pero sí considerábamos necesario hacer pequeñas aclaraciones de algunos niveles en que se dividió dicha grámatica. En la imagen 1 se pueden apreciar varios niveles intermedios de esta:

1. <comp>, que representa las operaciones donde se comparan elementos.
2. <arith>, que representa la operaciones de suma y resta.
3. <term>, para la multiplicación y división.

4. <factor>, como representación de los operadores unarios isvoid, opuesto y new.
5. <element> para las expresiones entre paréntesis, los *block* y las llamadas a funciones.
6. <atom> como el nivel más básico, donde se excuentran los números, ids, las expresiones booleanas y los string.

Estos niveles fueron orientados de esta forma en función de lo que el manual de Cool refería en cuanto a la precedencia de los operadores.

Lexer y Parser

Para tokenizar la entrada se utilizó una herramienta bastante práctica: PLY. Esta no es más que una impementación en python de las herramientas de parsing lex y yacc. Mediante el módulo *lex* nos provee de un analizador léxico ya implementado.

Para utilizarlo se definieron una serie de reglas para que el tokenizador trabajara en las cadenas de entrada y nos devolviera los tokens a gusto. Para ello se trabajó fundamentalmente con el módulo *re* de python que permite definir expresiones regulares. En el archivo *token_rules* se pueden observar dichas reglas.

El parse de *ycc* en cambio no fue el que utilizamos; decidimos seguir con nuestra implementación de proyectos pasados, la cual se puede apreciar en el archivo $shift\_reduce\_parsers$. En este caso utilizamos un parser de tipo LALR(1) por ser el de mayor alcance y el más acorde a la gramática de COOL.

Hasta este punto el trabajo efectuado puede reutilizarse para el proyecto final, sólo deben hacerse revisiones para comprobar que se ajuste a las siguientes fases del compilador y corregir errores. Antes de finalizar la segunda semana de septiembre dichas correcciones ya deben estar completas.

Análisis semántico

Para el proyecto de inferencia de tipos se implementaron 3 recorridos por el AST auxiliándonos del patrón `visitor` visto en clases. El primero de estos se encuentra en el archivo src/tset_builder.py y tiene como objetivo construir una estructura que permita almacenar para cada variable el conjunto de tipos que esta puede tener.

Esta estructura (a la que llamaremos `Tset`) es similar al `Scope`, de hecho nuestra primera intención fue construirla a partir del mismo, pero como los hijos de un scope no se sabe a qué nodo pertenecen, tuvimos que descartar esta idea, pues para las siguientes etapas del proceso esta información es imprescindible.

Cómo se guardan las variables y qué tipos se le asignan? Sencillo, se visitan los nodos donde se declaran variables (`AttrDeclarationNode`, `FuncDeclarationNode`, `LetNode`, `CaseNode`) y se le asigna a cada una el conjunto que contiene el tipo declarado, si el mismo es `AUTO_TYPE` el conjunto que le corresponde es el que contiene todos los tipos existentes.

```
<comp> := <comp> < <arith>
<comp> := <comp> = <arith>
<comp> := <comp> <= <arith>
<comp> := <arith>
<arith> := <arith> + <term>
<arith> := <arith> - <term>
<arith> := <term>
<term> := <term> * <factor>
<term> := <term> / <factor>
<term> := <factor>
<factor> := isvoid <element>
<factor> := ~ <element>
<factor> := new id
<factor> := <element>
<element> := ( <expr> )
<element> := { <block> }
<element> := <element> . <func-call>
<element> := <element> @ id . <func-call>
<element> := <func-call>
<element> := <atom>
<atom> := int
<atom> := id
<atom> := true
<atom> := false
<atom> := string
```

Figura 1. Fragmento de la gramática de Cool.

El segundo recorrido se encarga de reducir los conjuntos de tipos asociados a las variables y se ejecuta mientras existan cambios entre los tipos devueltos en la última pasada y la actual.

Las variables que antes tenían tipo `AUTO_TYPE` ahora cambiaron su tipo, por lo que hace falta hacer otra pasada chequeando tipos, para esto repetimos las 3 pasadas vistas en clases: `TypeCollector`, `TypeBuilder` y `TypeChecker`.

Ahora, esta implementación aunque nos es útil resulta perfectible. En esta, por cada recorrido se modificaba un mismo contexto sin crear un nuevo AST. Para esta fase pretendemos eliminar el contexto como una entrada aparte, y llevar en cambio un ast anotado, asignándo a este en cada pasada las propiedades necesarias, o creando un nuevo ast de sólo lectura con cada pasada (Dado que en python no se pueden aprovechar completamente las ventajas que puede ofrecer la segunda opción en un lenguaje como C#, lo más probable es que nos quedemos con al primera idea sin crear un nuevo ast con cada visitor). Estos cambios serán efecutados debido a lo complicado y engorroso que puede resultar nuestra propuesta inicial; consideramos que esta nueva idea, a parte de se la que usan los compiladores hoy en día resulta más elegante, clara y permite modificar con mayor facilidad cualquier parte del código sin trabas.

Dichos cambios se pretenden efectuar antes de culminada la primera semana de octubre, de modo que nos quede un margen de tiempo mayor para la parte final del proyecto que es la más trabajosa.

Generación de código

La generación de código es una de las fases más complicadas del compilador.

Una vez llegado este punto el código que se analiza es compilable, válido en Cool, no debe lanzar ningún error. La entrada a nuestro generador es un árbol de sintaxis abstracta anotado y la salida un archivo que contenga código MIPS equivalente a las instruccioens de entrada.

Ahora, para esta nueva fase no pensamos generar MIPS directo, dado que por el salto tan grande que habría que dar, dada la complejidad de un lenguaje orientado a objetos como COOL, lo abstracto que resulta el árbol que se recibe y la simplicidad del árbol al que se desea llegar, podríamos obviar casos esquinados o perder información relevante. En cambio usaremos un lenguaje intermedio, todavía por definir. Este debe porveer un balance entre que sea relativamente fácil traducir de semántica a él y de él a MIPS, para que ninguna pendiente de bajada sea muy larga. En estos pasos se desenrrollan las expresiones, se verifica que el orden de los operadores sea correcto, etc. Finalmente se pasa a generar código, conviertiendo este AST de lenguaje intermedio resultante a un AST de MIPS y llevando luego este último a texto.

Esta última fase debe ser completada antes de la segunda quincena de diciembre, aunque cabe destacar que todas las fases se podrían implementar al mismo tiempo, cada una por un miembro distinto del equipo, una vez definida la jerarquía de los ASTs, dado lo desacoplado que resultan cada uno de estos módulos descritos.