Universidad de La Habana

Proyecto de Compilación

Tiger

Authors:

Laila González Fernández

Mario C. Muñiz Jiménez

C412

March 7, 2016

1 Introducción

Siguiendo las especificaciones encontradas en el documento `Tiger Language Reference Manual` y las aclaraciones incluidas en el documento `TIGER - Proyecto de Compilación (2015-2016)` implementamos un compilador para el lenguaje Tiger.

Para la primera etapa del desarrollo de esta aplicación, el análisis lexicográfico y semántico, empleamos ANTLR 3.4. Para el análisis semántico y la posterior generación de código diseñamos una jerarquía de clases que representan las componentes semánticas de Tiger. Para generar un ejecutable usamos `CIL` que es ejecutado por la máquina virtual de `C#`.

Además diseñamos una jerarquía de tipos que incluye los tipos: `int`, `string`, `void`, `nil`, `array`, `record` y `error`. Igualmente se implementaron dos clases (`VariableInfo` y `FunctionInfo`) encargadas, entre otras cosas, de mantener enlazadas distintas referencias a una misma variable en el programa a un mismo espacio de memoria.

También se implementaron las clases `Scope` y `Report` de modo similar al visto en clases durante el curso de Compilación.

2 Análisis Lexicográfico y Sintáctico

El principal reto al que nos enfrentamos durante la confección de la gramática que utilizaríamos fue eliminar los prefijos comunes presentes en la gramática inicial de Tiger. Para eliminar este problema utilizamos los algoritmos de eliminación de recursividad izquierda inmediata y prefijos comunes estudiados en clase durante el curso de Compilación.

Con el fin de unificar el sistema de manejo de errores de esta fase (lanzamiento de una excepción de tipo `RecognitionException`) con el nuestro se expandieron las clases `TigerLexer.cs` y `TigerParser.cs` (usando el hecho de que son clases parciales) sobrescribiendo el método `ReportError`.

Existe una correspondencia entre los nombres de los nodos ficticios de nuestra gramática y los de nuestra jerarquía. De este modo puede implementarse el TreeAdaptor sin necesidad de un `switch` o un bloque `if` de grandes dimensiones, utilizando la clase `System.Reflection` presente en `C#`.

3 Análisis Semántico

La principal dificultad durante la fase de análisis semántico fue en la detección de circularidades en las declaraciones de alias y arrays.

Primeramente tanto para los bloques de declaraciones de tipos como los de funciones se separó el chequeo semántico en dos partes. En una primera pasada se analizan las partes izquierdas de las declaraciones y se agregan a los diccionarios de tipos o funciones según sea el caso. En un segundo momento se realiza el chequeo semántico de las partes derechas de las declaraciones. Esto permite que las declaraciones de tipos y funciones en un mismo bloque sean mutuamente recursivas.

Para solucionar este problema implementamos la estructura de Datos `DisjointSet` y explotamos sus peculiaridades. En cada bloque de declaraciones de tipo creamos un `DisjointSet` para cada uno de los tipos que intervienen en él. Al procesar cada declaración de un tipo alias o array se procede a unir (del modo habitual para los `DisjointSets`) los conjuntos correspondientes a ambos tipos involucrados en la declaración, representando de este modo la existencia de una dependencia entre estos.

Si en el momento de realizarse esta unión nos percatamos de que ambos conjuntos se encuentran ya unidos, hemos detectado una declaración circular.

De este modo e implementando una versión del `Path Compression` sobre `DisjointSets` se consigue eliminar las cadenas `a alias de b alias de c alias d` conviertiéndos en `a alias de d`, `b alias de d` y `c alias d` haciendo que las búsquedas posteriores del tipo real de un alias sean `O(1)`.

Una vez resueltos el resto de los tipos, se analizan los tipos Record.

Otro detalle que requirió nuestra atención fue el chequeo semántico de los nodos que representan un `break`. Para verificar la correctitud de la colocación del `break` se verifica que esté contenido en un `IBreakableNode` (interfaz implementada por los nodos `while` y `for`). Además se establece como `void` el tipo de retorno de todos las secuencias de expresiones entre el nodo `break` y el `IBreakableNode`.

4 Generación de código

Una primera aproximación para resolver los que a nuestro criterio eran los principales problemas a enfrentar durante esta fase: el anidamiento de funciones y la visibilidad y el ocultamiento de variables, fue representar cada `let` y cada definición de función

o lo que es lo mismo, los `scopes` generados por estos, como una nueva clase que hereda de la clase del `let` o definición de función que lo contiene. Nótese que, al estar cada uno de estos scopes en una clase distinta, estos pueden definir variables con el mismo nombre, y a la vez pueden acceder a la variables definidas en sus scopes padres.

Sin embargo, durante la implementación de esta solución, nos encontramos con el problema de que no puede usarse Reflection para encontrar el campo de un tipo cuya creación no ha sido terminada, ni se puede añadir campos a un tipo que ya ha sido creado.

Esto implica que en el siguiente código. Al procesar la línea señalada (dado que no se ha concluido la creación de la clase que reprsenta al primer let, no se puede acceder al campo `a` de este).

```
let
      var a := 5
      let
            var b := 7
      in
            a       <----------
      end
      a
in
end
```

Para dar solución a este problema se hacía necesario almacenar en cada clase un diccionario con todas las variables definidas en él y sus antecesores lo cual resultaba costoso.

La alternativa usada consiste en declarar todas las variables y funciones de manera estática, asegurando la unicidad del nombre de la declaración precediendo el nombre simple por el nombre del scope que contiene la variable que tambièn es único. Esto último se logra con un contador de `scopes`.

Otro aspecto a tener en cuenta era la posible recursividad en el llamado a funciones. Para asegurar que los valores de las variables locales antes y después de un llamado recursivo sean los mismos, se almacenan en la pila todos los valores que se encuentran entre el llamado a la función y su declaración antes de realizar el llamado y se sacan de esta una vez concluido el llamado para restablecer los valores de las variables.