Memoria del compilador: Yogur

Jorge Osés Grijalba y Álvaro Rodríguez García

Este documento detalla la sintaxis y la estructura del compilador del lenguaje de programación **Yogur**.

Características del lenguaje

Estructura de archivo

Los archivos de Yogur pueden contener código fuera de ámbito. Este es, de hecho, casi imprescindible, porque la ejecución comenzará por la primera línea de código del archivo. Por tanto, la ejecución de un archivo que contenga tan solo funciones o clases no va a hacer nada.

Identificadores y ámbitos de definición

- Los delimitadores serán los saltos de línea.
- Declaración de variables simples:

  ```
  var nombre: Tipo
  ```

- Declaración de arrays:

  ```
  var nombre: Tipo[longitud]
  ```

  Donde longitud es un entero.

- La delimitación de los bloques se hará al estilo c, abriéndolos con `{` y cerrándolos con `}`. Las instrucciones `if`, `while`, `for` ... abrirán bloques obligatoriamente.
- Las funciones no contarán con una cláusula return como tal sino que si devuelven una variable. Esta se especificará en la cabecera de la función (`argR` en el siguiente ejemplo). Se declararán de la siguiente manera:

  ```
  def nombre (arg1 : Tipo, arg2 : Tipo) -> argR : Tipo {
      ...
  }
  ```

  Las funciones solo pueden devolver tipos básicos: enteros y booleanos. Sin embargo, pueden aceptar cualquier objeto. Los tipos base y los arrays se pasarán por valor, pero las clases se pasarán por referencia.

- Para los procedimientos, la sintaxis será la misma pero sin valor de retorno. La sintaxis de clases será la siguiente:

```
class Clase {
    var a : Tipo
    ...
    def fun(...) -> argR : Tipo {
        ...
    }
}
```

Dentro de una función de clase, se podrá acceder a la clase que la declara con el identificador reservado `this`.

- Se accederá a los miembros de una clase y a sus funciones mediante el operador `.`, de la forma típica. Por supuesto, dentro de los métodos de clase se podrá utilizar el nombre de una variable o de otro método de clase sin necesidad de acceder mediante `.`.

Tipos básicos y operadores

- Los tipos predefinidos serán **Int** para los enteros (de 32 bits) y **Bool** para los booleanos.
- Los valores de los enteros pueden ser decimales o, si van precedidos por `0x`, hexadecimales.
- Los operadores infijos serán `+`, `*`, `-`, `/`, `and` o `&&`, `not` o `!`, `or` o `||`, `==`, `>=`, `>`, `<=`, `<`, y tendrán la asociatividad y prioridad que tienen en c. La asignación usará el símbolo `=`.
- También usaremos **false** y **true** como palabras reservadas.

Instrucciones ejecutables

- El acceso a los elementos de array se hará como es habitual con la notación `a[i]` para acceder al elemento `i` del array `a`, indexados desde 0.
- Las instrucciones condicionales tendrán la estructura usual:

```
if condition {
    // Instrucciones
} else if condition {
    // Instrucciones
} else {
    // Instrucciones
}
```

- Los bucles while también serán los habituales:

```
while condition {
    // Instrucciones
}
```

- La sintaxis de los for será, para un for desde `a` hasta `b`:

```
for i in a to b {
    // Instrucciones
}
```

- Las llamadas a funciones se realizarán de la siguiente manera:

```
fun(arg1, arg2, ...)
```

- Y las llamadas a procedimientos:

```
proc(arg1, arg2, ...)
```

Comentarios

Los comentarios serán estilo c, con `//` para un comentario de línea y `/* ... */` para un comentario de bloque.

Estructura del compilador

Proceso de compilación

La clase encargada de lanzar toda la compilación es la clase `yogur.Compiler`. Esta recibe dos argumentos al ser llamada: el primero es el path del archivo de entrada y el segundo, el path del archivo de salida. Si no recibe segundo argumento, inventa un nombre.

Organización de paquetes

El compilador está dividido en paquetes Java, que separan las distintas responsabilidades. Los paquetes más importantes son los siguientes:

- `yogur.utils`: contiene clases generales de utilidad que se usan a lo largo de todo el compilador. Por un lado, `CompilationException` es la excepción que se maneja en las distintas fases de compilación. Por otro lado, la clase `Log` cumple su propósito tradicional, de forma que podamos tener logs a lo largo del código que luego no se muestren si no lo deseamos.
- `yogur.jflex`: el encargado del análisis léxico. Aquí se encuentra la clase `YogurLex` generada por jflex.
- `yogur.cup`: se encarga del análisis sintáctico. Contiene las clases de cup `sym` y `YogurParser`.
- `yogur.tree`: este paquete contiene las clases del árbol abstracto. Tiene subpaquetes y subsubpaquetes, para diferenciar las declaraciones (paquete `declaration`), las expresiones (`expression`), los identificadores (`expression.identifier`), las instrucciones (`statement`) y los tipos (`type`). Entraremos en detalle más abajo.
- `yogur.ididentification`: tiene las clases necesarias para identificación de identificadores.
- `yogur.typeanalysis`: contiene las clases que utiliza el analizador de tipos.
- `yogur.codegen`: sus clases se encargan de la generación de código.

Árbol abstracto

Todos los nodos de nuestro árbol abstracto heredan de la clase `AbstractTreeNode`. Esta clase contiene algunos atributos que debe poseer cualquier tipo de nodo (por ejemplo, la línea y columna de declaración). Lo que es más importante, implementa la interfaz `AbstractTreeNodeInterface`, que define los métodos que deben implementarse a lo largo de todo el árbol. Las distintas fases de compilación llamarán a un método de esta clase para hacer su trabajo: de esta forma, son los propios nodos los que saben identificar sus identificadores, darse un tipo, y generar su código, de forma que todo está bien diferenciado.

Además, la raíz del árbol va a ser un nodo especial de la clase `Program`. Ahora, para el resto de nodos, tenemos una estructura de clases abstractas e interfaces que especifican las propiedades de cada uno: `Declaration`, `Expression`, `Identifier`, `Statement` ... De esta manera, cada nodo conforma a al menos una interfaz/clase abstracta de las anteriores, y así podemos tratarlos de forma abstracta desde los demás nodos.

Estructura de las fases de compilación

Se ha tratado de mantener las distintas fases de compilación diferenciadas en lo posible. Más allá de los analizadores léxico

y sintáctico, cuyo funcionamiento es el "obligado" por jflex y cup, nos quedan tres fases tras la construcción del árbol abstracto: identificación de identificadores, análisis de tipos, y generación de código.

Identificación de identificadores

Esta fase se lleva a cabo en cada nodo usando el método:

```
void performIdentifierAnalysis(IdentifierTable table) throws CompilationException;
```

Un objeto de la clase `IdentifierTable` se encarga de la gestión de la tabla de identificadores. Está implementado mediante un stack de mapas, donde cada mapa asocia una declaración (una referencia al árbol) a su identificador. Así, cada mapa del stack representa un nivel de anidamiento.

Esta fase de identificación guarda, en los nodos que lo necesitan, información sobre su declaración (normalmente una referencia), de manera que cuando ya no exista la tabla de identificadores se pueda recuperar tal información.

Análisis de tipos

En este caso, hay dos métodos en los nodos:

```
MetaType analyzeType() throws CompilationException;
MetaType performTypeAnalysis() throws CompilationException;
```

Para el primero de ellos, se usa siempre una implementación por defecto, que guarda en el nodo el tipo resultante del análisis. Aquí los tipos son "metatipos" (interfaz `MetaType`), que incluyen los tipos declarables pero también otros tipos útiles para la compilación como `void` o un tipo función. Estos no pueden declararse.

La fase de análisis de tipos se encarga también de asociar los accesos mediante `.` a su correspondiente declaración como atributo en una clase, de manera que luego sea sencillo recuperar esta información.

Generación de código

Se divide en dos subfases. La primera, de asignación de memoria, utiliza el método:

```
void performMemoryAssignment(IntegerReference currentOffset,
                             IntegerReference nestingDepth);
```

donde un `IntegerReference` es tan solo un wrapper de un entero que se puede pasar por referencia. Esta fase de asignación de memoria guarda, en los nodos que lo necesitan, información sobre la profundidad de anidamiento en funciones, los offsets, etc.

La segunda subfase es la generación de código propiamente dicha, mediante la familia de funciones del estilo de:

```
public void generateCode(PMachineOutputStream stream) throws IOException
```

que se corresponde con las funciones `code`, `codeL`, `codeR`, `codeI`, `codeA` ... cada una de ellas en el nodo que las necesita. La generación de código sigue los esquemas estándares, excepto en alguna instrucción como el `for` en la que es necesario crear un esquema propio.

Esta fase utiliza un `PMachineOutputStream`, una subclase de `FileWriter` que va almacenando las instrucciones generadas hasta que se escriben. Esta estructura tiene la ventaja de que a lo largo del compilador podemos tratar las etiquetas como tal, y dejamos al stream la tarea de, en el momento de escribir, asignar a cada etiqueta su dirección absoluta.