Guía Práctica 3 - MII 779
Análisis Semántico y Representación Intermedia en Java+ANTLR
Desarrollo de Lenguajes Orientados a Objeto y Compiladores Avanzados

Profesor: Ricardo Soto

Ejercicio 1: Instalación del analizador semántico para el lenguaje Mile++

- Descargue el archivo `milePP-semantic.zip` desde `http://www.inf.ucv.cl/~rsoto/cursos/MII779/milePP-semantic.zip`
- Descomprima y copie en su workspace.
- Cree un proyecto seleccionando como fuente del proyecto la carpeta recientemente descomprimida.

Note que el proyecto contiene dos nuevos iteradores de árboles: `MilePPTreeParser.g` y `MilePPTreeParser2ndPass.g`. Un analizador semántico para un lenguaje orientado a objetos requiere de dos exploraciones. A modo de ejemplo, considere el siguiente caso en relación a la validación de declaración de variables.

```
class test1(){
  test2 a;
}

class test2(){
  int b;
}
```

En la primera exploración, el analizador semántico comienza verificando el tipo de la variable `a`, el cual es `test2`. Sin embargo, no puede saber si la clase `test2` existe dado que no ha explorado aún esa clase. Por lo tanto, se procede a realizar el análisis semántico en dos "pasadas". En la primera se almacena toda la información referente a las clases y en la segunda se verifican los tipos.

Ejercicio 2: Comprenda la representación intermedia del lenguaje Mile++.

- La representación intermedia almacena la información del programa fuente con el fin de facilitar la validación semántica, la optimización de código y la generación del mismo. La representación intermedia de un programa en Mile++ se administrará en el paquete `src->cl.ucv.inf.mileppcompiler->compilers->programInfo`.

La presente representación intermedia mantiene un estilo orientado a objetos con el fin de representar fielmente la estructura del lenguaje Mile++. De esta forma, se consigue una estructura jerárquica organizada y más fácil de manipular.

- El paquete `programInfo` esta compuesto por:
  - `mileProgram` contiene y administra la información de un programa en Mile++.

- `mileClass` contiene y administra la información de una clase en Mile++.
- `mileAttribute` contiene y administra la información de un atributo en Mile++.

Ejercicio 3: Comprenda la nueva clase `SemanticInspector.java`

- Diríjase a `src->cl.ucv.inf.mileppcompiler->compilers->SemanticInspector.java`. Esta clase permitirá en conjunto a los iteradores de AST verificar la semántica del lenguaje Mile++.
- La clase SemanticInspector.java contiene 4 métodos:
  - `addClass` agrega clases a la representacón intermedia. Si la variable ya existe, despliega un mensaje de error.
  - `checkExtends` verifica si una clase se extiende a si misma.
  - `checkObject` verifica si el tipo de un objeto existe.
  - `semanticError` se encarga de darle el formato al mensaje de error.

Ejercicio 4: Pruebe el analizador semántico

- Ejecute la clase `Tool` utilizando como programa fuente el archivo `examples/test1.mile`. La ejecución debería entregar 2 errores semánticos.

Ejercicio 5: Complete la representación intermedia

- Implemente las reglas faltantes de los iteradores de AST.
- Almacene la informacíon faltante en la representación intermedia. Agregue las clases al paquete `programInfo` que estime conveniente.

Ejercicio 6: Implemente las validaciones semánticas propias de un lenguaje orientado a objetos

- Validar la existencia de la superclase invocada.
- Un acceso (`a.getObjeto()`) no puede llamar a un objeto o método inexistente.
- Evitar que una clase posea como atributo un objeto de su propia clase.
- Evitar composiciones infinitas, por ej:

```
class Test1{          class Test2{          class Test3{
  Test2 b;              Test3 a;              Test1 a;
}                     }                     }
```

Ejercicio 7: Implemente y pruebe el analizador semántico de su lenguaje.