Departamento de Ciência da Computação - UFJF
DCC045 - Teoria dos compiladores

Analisador Semântico para a Linguagem *lang*

Alunos:

Daniel Souza Ferreira - 201665519B

Matheus de Oliveira Carvalho - 201665568C

Agosto / 2021

1 Introdução

Neste trabalho desenvolvemos o analisador semântico para a criação de um compilador para a linguagem *"lang"*.

É possível a existência de programas que sejam sintaticamente corretos porém não fazem sentido, tendo comportamento inesperado. Alguns requisitos para determinar se o programa irá executar operações ilegais da máquina é verificar se ocorre divisão por zero, acesso inválido da memória, referência a variáveis ou funções inexistentes, etc. O papel do analisador semântico é procurar esses erros que não são identificados pelo analisador sintático e emitir uma mensagem de erro caso os encontre.

2 Características do Projeto

Para o desenvolvimento dessa parte do compilador utilizamos a árvore AST gerada no trabalho anterior. Para conseguirmos realizar a análise em nível semântico foi necessário a criação de classes que representam os tipos de dados de uma linguagem, sendo feitos usando a representando a forma semântica e não a forma sintática, como no trabalho anterior. Vale dizer que essas classes foram criadas seguindo o padrão *Singleton* visto na aula da disciplina.

Também foi necessário a criação das classes *TestVisitorSemantic.java* e *VisitorSemantic.java*, com o objetivo de realizar a análise semântica dos nós da AST. Essas classes foram criadas usando a *TestVisitorInterpretator.java* e *VisitorInterpretator.java*, respectivamente, como base para suas implementações uma vez que elas seguiam princípios em comum.

2.1 Novos Arquivos Presentes no Trabalho

- Arquivos presentes na pasta *type*
  - SType.java
  - STypeArray.java
  - STypeBool.java
  - STypeChar.java
  - STypeData.java
  - STypeFloat.java
  - STypeFunc.java
  - STypeInt.java

  - STypeNull.java
  - STypeVar.java

2.2 Novos Arquivos Presentes no Trabalho

- VisitorSemantic.java
- TestVisitorSemantic.java

3 Decisões de Projeto

3.1 Estruturas

- **stk:** Pilha de SType responsável por aramazenar todos os tipos que são análisados no decorrer na execução do programa em si.
- **tempFunc:** Variável do tipo STypeFunc responsável por armazenar temporariamente a função que está sendo acessada.
- **env:** Hashmap para armazenar os tipos de variáveis. Contém Strings representado o id do tipo funcionando como key e variáveis do tipo SType funcionando como value.
- **funcs:** HashMap para armazenar as funções do programa. Contém Strings representado o id do tipo funcionando como key e uma ArrayList do tipo STypeFunc funcionando como value. Essa ArrayList é criada com a finalidade de armazenar funções sobrecarregadas.
- **typeInt, typeFloat, typeBool, typeChar, typeNull, typeVar:** Variáveis criadas com a finalidade de representar a instância de cada tipo básico.
- **typesData:** HashMap para armazenar as estruturas Datas do programa. Contém Strings representado o id da estrutura Data funcionando como key e uma variável do tipo STypeData funcionando como value.

3.2 Decisões

- O programa se encerra ao encontrar um erro.
- Somente valores do tipo inteiro serão lidas pela operação *Read*. Ao encontrar um valor do tipo variável o mesmo será "convertido"para inteiro utilizando seu id original.

- Todas operações de tipo aritmético verificam o tipo dos operadores. O tipo Float sempre irá sobrepor o tipo Int e somente os tipos Int e Float são executados.
- O comando New, aceita apenas tipos compostos como arrays, matrizes e structs (Data).
- O comando Mod(modulo) só aceita ser usado com variáveis do tipo Int.
- O comando Read só é usado com variáveis do tipo Int.
- Foi necessário a criação de classes auxiliares para a execução correta de alguns comandos. Essas classes foram criadas para realizar algumas recursividades necessária e para evitar a utilização de código repetido.

4 Execução do Programa

Para a execução do programa foi criado um makefile que possui os seguintes comandos:

- **make** - utilizado para fazer a limpeza dos arquivos .class gerados na pasta 'lang' e depois realizar a compilação de todos os arquivos .java.
- **make clean** - utilizado para fazer a limpeza dos arquivos .class gerados na pasta 'lang'.
- **make execute** - utilizado para realizar a execução da bateria de testes.

A bateria de testes gerada pelo make execute é dos arquivos da pasta 'certa' da pasta 'semantica'. Para realizar os testes dos arquivos da pasta 'errado' deve-se alterar o valor da variável **okSrcs** na classe **TestVisitorSemantic.java** para **"testes/semantica/errado/"**.