Analisador Semântico
DCC045 - Teoria dos Compiladores

André Felipe de Souza Mota
20166515B

Eugenio Belizário Ribeiro Faria
201665507B

Novembro de 2020

1 Introdução

Uma vez produzida a árvore sintática do código, coerente com a gramática da linguagem, o analisador semântico será responsável por fazer a verificação de tipos, identificar possíveis erros semânticos e problemas indesejados, como a inconsistência entre tipos em uma operação, e verificar a unicidade na declaração de variáveis, funções e tipos de dados.

2 Categorização de tipos

Para melhor compreensão das seções seguintes, faz-se necessário explicitar a divisão conceitual adotada ao longo do trabalho: a categorização entre tipos primitivos e tipos compostos.

Tipos primitivos são formados por *Int*, *Float* e *Bool*, enquanto tipos compostos são, em geral, vetores e registros.

3 Regras de projeto

Para a implementação do trabalho, levou-se sem consideração a definição técnica do mesmo e especificação da linguagem Lang, todavia haviam casos não contemplados ou abertos para serem definidos durante a implementação. Desta forma, fez-se necessário a definição de algumas regras de projeto, são elas:

- Ao se encontrar um erro, a execução é finalizada e uma mensagem é impressa.
- A atribuição de *null* à uma variável de tipo primitivo não é uma operação válida.

- A verificação de unicidade também se estende a tipos de dados, não sendo possível a criação de atributos com o mesmo nome.
- O retorno em uma função, quando existente, deverá ser garantido, ou seja, em todas as possibilidades de execução deverá ser possível executar o *return*, incluindo situações que o retorno esteja encapsulado por uma ou mais estruturas condicionais.
- A verificação de tipos, para a operação de módulo, garante que só serão aceitos operandos do tipo inteiro.
- Para a chamada de funções com múltiplas possibilidades de retorno, contemplada na documentação como "ID (EXPS) [EXP]", a especificação de qual retorno se deseja obter, utilizando-se colchetes, deverá ser feita com um literal inteiro.
- Optou-se por considerar o escopo de estruturas condicionais, como *if* e *iterate*, como global.

4 Implementação

O analisador semântico utiliza diretamente a AST gerada pelo analisador sintático e, para sua implementação, adotou-se o padrão de projeto utilizado no interpretador, o *visitor*. Seu código pode ser visto no arquivo VisitorStatic.java e a metodologia de implementação foi similar a apresentada em sala de aula, com algumas adaptações.

4.1 Escopo

A implementação do escopo não requisitou a criação de uma classe auxiliar, ao invés disso foi utilizada uma pilha de escopo. Seu funcionamento consiste em, quando chamada uma função, insere-se as variáveis passadas como argumento em uma pilha e, conforme novas declarações vão sendo feitas durante sua execução, novas inserções são feitas na pilha. Ao fim da execução da função, as variáveis locais são removidas da pilha.

4.2 Verificação de tipos

A implementação da verificação de tipos foi feita através da criação de classes auxiliares que representam cada um dos tipos possíveis na linguagem *Lang*, sendo eles *Int*, *Float*, *Bool*, *arrays* e *datas*, além de representações para funções. Dessa forma, sempre que encontramos um dado tipo, ele é empilhado em uma pilha de tipos para posterior comparação.

5 Execução do código

Para a execução o analisador semântico, basta executar o seguinte comando na pasta do código fonte:

```
$ make file=caminhoArquivo
```

Sendo "caminhoArquivo" o caminho da pasta raiz do trabalho até o arquivo de execução, por exemplo:

```
$ make file=testes/sintaxe/certo/attrEQ.lan
```

O retorno será o resultado um OK, caso não exista problemas no código, ou o erro encontrado, caso contrário.

Caso se deseje executar o interpretador após a analise semântica, basta rodar o seguinte comando:

```
$ make file=caminhoArquivo interpretar=true
```

Além dos comandos básicos apresentados anteriormente, alguns complementarem foram implementados no *makefile* com o intuito de facilitar mudanças e testes, sendo eles:

```
$ make clean
```

Apaga todos os arquivos do tipo ".class" da pasta "lang".

```
$ make lexical
```

Apaga o "Lexical.java" gerado pelo JFlex anteriormente e recria um novo.