Relatório - Analisador Sintático e Interpretação

04 de Junho de 2023

Nome Completo	**Matrícula**
Lívia Pereira Ozório	201835011
Patrick Canto de Carvalho	201935026

Sumário

1 Introdução

Este relatório tem como objetivo explicitar os pontos mais relevantes para a compreensão do processo de desenvolvimento de um analisador sintático e interpretador para a linguagem *lang*, incluindo as decisões de projeto como as estratégias adotadas, a definição e descrição de ferramentas, estruturas de dados auxiliares e representação da árvore de sintaxe abstrata utilizada.

2 Ferramentas e estrutura básica

Para o desenvolvimento deste trabalho, foi escolhida a ferramenta ANTLR para a linguagem Java para a geração do analisador sintático. ANTLR é um gerador de analisadores *top-down* que implementa a estratégia chamada *LL(*)*. É importante notar que a ferramenta mencionada une a análise léxica à sintática. Sendo assim, as regras léxicas da linguagem para a sintaxe da ferramenta e adicionadas ao arquivo que define a gramática, pois anteriormente estava sendo usado o JFlex.

Outra ferramenta utilizada é a linguagem *dot*, para gerar a visualização gráfica da árvore de sintaxe abstrata.

Dessa forma, o artefato principal consiste em um programa escrito na linguagem Java, acoplado ao analisador gerado pelo ANTLR com base nas regras especificadas, responsável por receber informações do arquivo de entrada e coordenar a execução da análise e a interpretação da linguagem, exibindo o resultado da execução no terminal de comandos e também gerando um arquivo *dot*.

3 Representação da AST e estruturas de dados

O produto da análise sintática é uma árvore de sintaxe abstrata (AST). Portanto foi necessário desenvolver uma representação interna adequada. Esta consiste em classes que representam nós, conectados por meio de relações sintáticas.

As classes implementadas são as que seguem:

- Node: classe pai de todos os outros as outras classes que representam nós da árvore;
- Prog: é o nó de entrada para a execução do programa;
- Data: representa a declaração de uma estrutura de dados;
- Func: representa a declaração de uma função;
- Add, Div, Mul, Sub e Rest: representam operações binárias de adição, divisão, multiplicação e subtração e módulo, respectivamente;
- Diff, GreaterThan, LessThan, Eq: representam as operações binárias de comparação;
- BinOp: classe pai daquelas que representam operações binárias;
- SubUni: representa o sinal de subtração unário;
- And e Neg: representam as operações lógicas AND e NOT;
- Bool, Int, Char, FloatAst e Null: representam os tipos básicos de dados e o valor *null*;
- ID: representa um identificador;

- CallFunction, CallFunctionVet, If, Attr, Iterate, ReturnCMD, Print, Read: representam comandos de chamada de função, chamada de função com captura do vetor de retorno, estrutura condicional, atribuição, laço de repetição, retorno de função, impressão e leitura de dados, respectivamente;
- Expr, Type, LValue, New: representam diferentes expressões, todas são da classe Expr.
- Decl: representa uma declaração de atributo de uma estrutura de dados (*data*);
- Param: representa um parâmetro de função;
- CmdList FuncList, DeclList, DataList, TypeList, ExprList, LValueList, ParamsList: representam sequências de elementos. Embora não existam regras da gramática que correspondam diretamente a estas estruturas, elas auxiliam na organização e clareza da gramática e da AST resultante, permitindo uma interpretação iterativa destas sequências, em oposição à recursiva;
- DataInstance: não corresponde a nenhum nó na árvore, apenas representa uma instância de um tipo de dados definido no código *lang* criada pelo comando *new*.
- IdGenerator: não corresponde a nenhum nó na árvore. Auxilia na identificação de cada nó da árvore por meio da geração de um inteiro (*id*) único, possibilitando posteriormente a criação da visualização desta no formato *graphviz*.

4 Interpretação

Cada classe correspondente a um nó da AST possui um método *tryInterpret*, responsável por interpretar o programa codificado nas suas subárvores filhas e simular o comportamento desejado. Este método recebe uma pilha de *hashmaps*, uma lista de funções, uma lista de estruturas de dados e uma pilha de lista de objetos. Ele é chamado do programa principal a partir da classe Prog, que executa recursivamente o método *tryInterpret* dos nós filhos, que por sua vez executa o dos seus filhos até chegar às folhas. Caso haja um erro durante a chamada desta função, o analisador retorna uma mensagem informando a localização do erro.

A pilha de *hashmaps* contém as varáveis presentes em cada escopo dentro de uma execução do programa. Sendo empilhado um novo *hashmap* toda vez que uma função é chamada, e desempilhado após o seu término. Ou seja, o topo da pilha sempre contém as variáveis pertencentes ao escopo da função corrente em execução, sendo possível para esta consultar apenas a ele.

A lista de funções e a lista de estruturas de dados permitem que as definições de funções estruturas de dados sejam consultados em qualquer ponto do programa, independente de escopo.

A pilha de lista de objetos que aparece ao fim é responsável por armazenar os retornos das funções conforme são terminadadas (para aquelas que possuem retorno). Quando os retornos são lidos, o topo é desempilhado.

As funções *tryInterpret* de cada nó podem ser muito diferentes, visto que cada nó possui estruturas e expressa relações sintáticas diferentes, bem como comportamentos diferentes.

5 Geração de código *dot*

Pode ser útil visualizar graficamente a estrutura da AST gerada na análise sintática, sendo assim, além da função *tryInterpret*, cada classe correspondente a um nó possui uma função

dotString, responsável por gerar código na linguagem *dot*. Esta linguagem é utilizada para geração de visualização de grafos pelo software *graphviz*.

Cada nó aciona recursivamente a função *dotString* dos seus nós filhos, da mesma forma que a função *tryInterpret*. Para que cada nó possa ser referenciado de forma não ambígua dentro do código, todos recebem um *id* único gerado por meio da classe IdGenerator.

6 Como compilar

Para compilar e executar usando o programa Make, basta executar o seguinte comando no diretório do projeto:

```
make run <nome do arquivo lang>
```

Alternativamente, executar os seguintes comandos para compilar :

```
javac -cp .:antlr-4.8-complete.jar Interpretador.java
java -jar antlr-4.8-complete.jar parser/lang.g4
```

E o seguinte para executar:

```
java -cp .:antlr-4.8-complete.jar Interpretador <nome do arquivo lang>
```

Para realizar os testes presentes na pasta testes/semantica/certo, executar:

```
./teste.sh
```

O programa deverá ter as suas saídas apresentadas no terminal. Além disso, é gerado um arquivo *ast.dot* para visualização da árvore de sintaxe abstrata.