TRABALHO PRÁTICO - INTERPRETADOR

Bruno Marcos Pinheiro da Silva
201565552AC

Seany Caroliny Oliveira Silva
201665566C

1. INTRODUÇÃO

Este documento visa descrever as principais características da implementação do Interpretador para a linguagem "*lang*", proposta na disciplina de Teoria dos Compiladores.

O interpretador desenvolvido é uma extensão da etapa anterior do trabalho, que resultou na geração de uma Parse Tree fornecida pela ferramenta ANTLR[1] após a implementação da análise sintática. Esta Parse Tree foi percorrida com a interface Visitor do próprio ANTLR para a criação de uma Árvore de Sintaxe Abstrata definida por nós e que é posteriormente utilizada em um Visitor customizado para a interpretação do código.

2. ESTRUTURA DE ARQUIVOS

O trabalho consiste na seguinte estrutura de arquivos:

- `lang/`
  - `ast/`
    - `SuperNode.java`
    - **`Arquivos referentes aos nós da AST`**
  - `parser/`
    - `TestParser.java`
    - `ParseAdaptor.java`
    - `FinalParser.java`
    - `CustomErrorListener.java`
    - **`BuildAstVisitor.java`**
    - **`InterpretAstVisitor.java`**
    - `(LangLexer/LangParser.java)`
  - `testes/...`
  - `LangCompiler.java`
  - `LangLexer.g4`
  - `LangParser.g4`
  - `Compile.bash`
  - `CompileAntlr.bash`

[1] https://www.antlr.org/

3. COMO EXECUTAR

Para usar o programa, basta executar as seguintes linhas no terminal de comando aberto na pasta do projeto:

- O .jar do ANTLR deve estar na pasta `.\lang` do projeto.
- Para compilar as classes Java e executar o programa, a partir da pasta `lang`:

```
javac -cp .:antlr-4.8-complete.jar ast/*.java parser/*.java LangCompiler.java -d .
java -classpath .:antlr-4.8-complete.jar lang.LangCompiler -bs
```

- Caso seja necessário criar as classes do ANTLR novamente, deve-se executar os seguintes comandos:

```
java -jar ./antlr-4.8-complete.jar -o ./parser/ LangLexer.g4
java -jar ./antlr-4.8-complete.jar -o ./parser/ LangParser.g4 -visitor
```

Os arquivos `compile.bash` e `compileAntlr.bash` serão incluídos no projeto para executar esses comandos automaticamente, considerando a presença do ANTLR (`antlr-4.8-complete.jar`) na pasta `lang`.

- A pasta `./testes` possui os testes previamente utilizados para o analisador sintático. Nem todos os arquivos são aceitos pelo interpretador.

4. DECISÕES DE PROJETO

4.1. Estratégia de Interpretação

Para a implementação do interpretador utilizamos o padrão Visitor, como apresentado durante as aulas.

A classe `FinalParser.java` foi alterada para realizar o Parse, construir a AST e interpretá-la.

4.2. Nós da Árvore de Sintaxe Abstrata

Para criarmos nossa AST, definimos nós relativos às construções relevantes da nossa linguagem. Alguns nós são utilizados somente para facilitar a construção e outros pelo visitor do ANTLR, não compondo a AST final. Os nós são resumidos na tabela a seguir:

Classe	Construção da Linguagem
Operações Binárias/Unárias	
OpSum	Soma: a + b
OpSub	Subtração: a + b
OpMul	Multiplicação: a * b
OpDiv	Divisão: a / b
OpMod	Resto: a % b
OpAnd	Conjunção: a && b
OpNot	Negação lógica: !a
OpEq	Igualdade: a == b
OpLess	Relacional: a < b
OpMin	Menos unário: -a
OpNotEq	Diferença: a != b
Comandos	
CmdAssign	Atribuição: a = b
CmdFunctionCall	Chamada de função com alocação dos retornos
CmdIf	if (exp) cmd if (exp) cmd else cmd
CmdIterate	iterate (exp) cmd
CmdList	Lista de comandos
CmdPrint	print exp
CmdRead	read lvalue
CmdReturn	return exp
Expressões/Literais/Referência	
ExpFunctionCall	Chamada de função com seleção do retorno

ExpNew	Alocação: new ...
LiteralBool	true ou false
LiteralChar	‘a’, ‘A’, ‘0’, ...
LiteralInt	0, 1, 2, ...
LiteralFloat	3.141526535, 1.0 e .12345
LiteralNull	null
LvalueID	ID
LvalueArray	lvalue[exp]
LvalueSelect	lvalue.ID

O nó menos intuitivo trata-se do **LvalueID**. Este nó possui o ID da variável/referência principal e uma lista de *seletores* que podem ser **LvalueArray** e **LvalueSelect.** Estes seletores são utilizados para acessar os endereços pertinentes relacionados ao ID do LvalueID. É sempre este o tipo do primeiro Lvalue encontrado enquanto a AST é percorrida na interpretação.

4.3. Visitor do ANTLR

A interface Visitor do ANTLR é utilizada para construção da AST. Com os nós definidos anteriormente, implementamos a interface base fornecida pela ferramenta na classe `BuildAstVisitor`, que fornece acesso às derivações de cada regra da gramática e a criação manual dos nós que compõem a AST, retornando um SuperNode.

Cada função neste visitor corresponde à uma das derivações da gramática. Dentro destas funções, temos o objeto de “contexto” para o nó da árvore de Parse fornecida pelo ANTLR correspondente à derivação corrente na gramática e, nestes contextos, temos funções para visitar os nós filhos da regra corrente.

4.4. Visitor para a Interpretação

O visitor implementado para a Interpretação segue o padrão apresentado durante as aulas e foi construído manualmente. Este foi implementado no arquivo `InterpretAstVisitor.java`. Temos funções `visit` para cada nó definido em nossa AST que realizam operações pertinentes às construções correspondentes da linguagem. Nos tópicos a seguir descrevemos as principais características para entendimento do interpretador.

- **`datas`**: um HashMap que salva os tipos de dados definidos pelo usuário, através de um id e a estrutura Data;
- `funcs`: um HashMap que salva as funções definidas pelo usuário, através de um id e a estrutura Func definida para a função;
- **`lvalues`**: a resolução de lvalues são realizados de forma recursiva. Um Lvalue possui seletores que podem representar um acesso à um array ([exp]) ou um acesso à propriedade (.id). No caso de acesso a arrays, acessamos o operando de acordo com a posição resolvida por "exp" e, no caso de propriedades, acessamos o valor armazenada no hashmap associado àquela posição através do ID do seletor.
- **`new`**: neste caso, verificamos se existe uma expressão de seleção. Neste caso, temos a definição de um Array, alocado em uma ArrayList e, caso contrário, a alocação de um novo tipo "Data", que é alocado em um HashMap com as propriedades do tipo específico.
- **`checkArithmeticType`**: função implementada para facilitar a conferência e conversão de tipos durante as operações aritméticas.

4.5. Dificuldades e possíveis problemas

Durante a implementação deste interpretador, nos deparamos com alguns problemas, dado que alguns programas estavam sintaticamente corretos mas não possuíam sentido, falhando na interpretação. A maioria destes problemas está relacionado com a falta de conferência de tipos de operações e retornos de função, assim como a ausência de mecanismos para lidar com parâmetros na função 'main'.

Quanto à dificuldades de implementação, a maior parte consistiu em mapear as construções disponíveis em Java para aquelas da nossa linguagem.

5. CONCLUSÃO

Com a conclusão do interpretador, foi possível ganhar mais familiaridade com o conceito de Árvores de Sintaxe Abstrata, assim como o padrão de projeto Visitor. Foi possível notar que somente a análise sintática não é suficiente para determinar se um programa é válido, dado que sua interpretação é passível de vários erros que deveriam ser previamente detectados.