Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação
DCC045 - Teoria dos Compiladores Semestre ERE 2020-1

Análise Semântica

Edson Lopes da Silva Junior 201635023

Vinicius Alberto Alves da Silva 201665558C

Professor: Leonardo Vieira dos Santos Reis

Relatório do trabalho prático Análise Semântica, parte integrante da avaliação da disciplina.

Juiz de Fora

Novembro de 2020

1 Introdução

Este relatório é do trabalho prático 4 da disciplina de Teoria dos Compiladores do Ensino Remoto Emergencial (ERE) 2020.1 e tem como objetivo descrever o processo da implementação de um Analisador Semântico para a linguagem *Lang*.

2 Metodologia Utilizada

Devido o padrão de projeto escolhido para implementação do Interpretador no projeto anterior foi o Visitor, a escolha para essa etapa é implementar um Analisador Semântico seguindo o mesmo padrão.

Para implementação do Visitor o Antlr fornece uma opção de atravessar á árvore de sintaxe abstrata, ele automaticamente gera as classes usadas na visita dos nós da AST ao passar o parâmetro -visitor ao gerar os arquivos. O objetivo final é analisar a árvore verificando informações sobre os tipos dos objetos e desta forma verificar ser o código-fonte fornecido está de acordo com as especificações da linguagem.

2.1 Organização do Código

Com relação a organização do código, a pasta Parser contém os arquivos gerados pela ferramenta Antlr: LangLexer.tokens, LangLexer.interp, Lang.interp, LangParser.java, LangBaseListener.java, LangLexer.java, LangListener.java. Estes são que estão responsáveis pela análise léxica e sintática. Nesta mesma pasta estão os arquivos LangVisitor.java e LangBaseVisitor.java que são as classes geradas pelo Antlr responsáveis por percorrer a AST.

Também na pasta Parser está o arquivo CreateASTFromParser.java que estende LangBaseVisitor<SuperNode> e implementa os métodos de visita de cada nó da AST. Por último, a classe LangAdaptor implementa a interface ParseAdaptor. A classe LangAdaptor é responsável por chamar o método ParseFile, este faz a leitura do arquivo a partir da classe LangLexer e é feito o parser com a classe LangParser.

Na pasta AST encontram-se as classes que representam cada nó da AST. Todas as classes herdam da superclasse abstrata SuperNode. Além disso todas as classes possuem a possibilidade de utilizar o método *accept* utilizado na execução do Visitor.

Na pasta Visitor esta a interface Visitable, a classe abstrata Visitor, a classe responsável pelo interpretador: InterpretVisitor e por último o TypeCheckVisitor, neste é realizado todo o processo de análise semântica.

Para a Análise Semântica foram criadas novas classes para cada tipo de dados, conforme modelo visto em aula, estas classes se encontram na pasta TypeCheck.

A classe TesteParser fornecida no trabalho anterior foi alterada para rodar apenas um teste, o arquivo para teste chama-se mytest.lan e encontra-se na pasta testes/sintaxe.

2.1.1 Como executar

Para executar basta executar o arquivo bash run.sh, ele contem o seguinte conteúdo:
#!/bin/bash

if [-z "$1"]; then antlr_jar="antlr-4.8-complete.jar"; else antlr_jar=$1; fi

java -jar ./$antlr_jar -o -visitor ./parser/ Lang.g4
javac -cp .:$antlr_jar ast/*.java parser/*.java TypeCheck/*.java LangCompiler.java -d .
java -classpath .:$antlr_jar lang.LangCompiler -bs

É possível trocar o endereço do jar do antlr 4.8 por outro de sua preferência, basta fornecer o caminho do .jar na chamada de execução do script, exemplo:
bash run.sh pasta1/pasta2/antlr-4.8-complete.jar

2.1.2 Teste exemplo

Um código-fonte na linguagem lang foi escrito no arquivo testes/sintaxe/mytest.lan , este arquivo contém o conteúdo descrito abaixo. Quando executada, nossa solução já executa automaticamente este teste. Conforme especificação da linguagem, os seguintes erros são fornecidos:

8,7 Operador:+ não se aplica aos tipos Float e Int
14, 7: Variável não declarada Quadrado
15, 9: Função f__Int não declarada
17, 3: Comando READ só funciona para números inteiros p

Programa mytest.lan

```
data Ponto {
x :: Float;
y :: Float;
}
f(x :: Float) : Float {
y = 2.3*x + 1;
return y;
}
main() {
p = new Ponto;
z = Quadrado;
p.x = f(10)[0];
print p.x;
read p.y;
print p.y;
}
```

2.1.3 Detalhes de implementação

Na gramática (arquivo Lang.g4) cada regra foi rotulada com um #nome_rotulo. Desta forma o antlr gera um método *visit* para o objeto correto de acordo com o rótulo. As regras que tem apenas uma derivação não precisam de ser rotuladas.

O arquivo gerado automaticamente pelo Antlr *LangBaseVisitor* contém a chamada dos métodos *visit*. Desta forma, foi implementada uma nova classe que estende *LangBaseVisitor* e subscreve seu métodos. Esta nova classe é chamada de *CreateASTFormParser*. A facilidade desta forma de implementação é que visitar um nó da árvore pode ser encarado como apenas chamar as regras definidas na gramática.

Também foi necessário definir como vamos realizar a análise semântica, a partir do Visitor, cada nó. Isso é feito na classe *TypeCheckVisitor*, de forma similar ao que foi apresentado na aula.

2.1.4 Estruturas de tipos utilizadas

As classes *LocalEnv* e *TyEnv* seguem a proposta apresentado pelo professor nas aulas e no exemplo.

Para as classes as classes *STyInt*, *STyFloat*, *STyChar*, *STyNULL*, *STyBool* e *STyErr* na pasta TypeCheck foi escolhido o padrão *singleton* de desenvolvimento. Desta forma, cada classe é instanciada uma única vez nos atributos de *TypeCheckVisitor*. Já as classes *STyData*, *STyArray* e *STyFunc* seguem o padrão normal de desenvolvimento, com seus construtores públicos, já que cada instância de um tipo desses é unica.

A classe STyData possui um identificador e um HashMap<String, SType> para guardar os múltiplos atributos que um tipo Data pode possuir. Para mapeamento de todos os tipos de dados heterogêneos de uma programa foi criado um *HashMap<String, STyData>* **datas**. Nesta estrutura o identificador do tipo Data é a chave que aponta para o próprio Objeto tipo data.

Para Armazenamento de logs de erro, foi criado uma variável
ArrayList<String> **logError** em que cada item da lista é uma mensagem de erro.

A variável $TyEnv < LocalEnv < SType >>$ **env** guardas as informações das funções declaradas no programa.

Já a variável *LocalEnv<SType>* **temp** é o objeto onde são armazenadas as variáveis declaradas no programa.

A pilha *Stack<SType>* **stk** guarda informações sobre os tipos dos objetos.

A variável *boolean* retCheck é uma forma de controle para o retorno de funções.

2.1.5 Estratégias utilizadas para definir a semântica da linguagem

Para possibilitar a sobrecarga de funções, optou-se por adicionar quais são os tipos dos parâmetros que implementam a função separados por dois *underscores* "__". Considere o seguinte declaração de função: $func(x :: Int, y :: Float)$, ao visitar a função utilizando o visitor é feito um processo adicional para extrair os tipos dos parâmetros para incluir no id final da função, resultando em "$func\_\_IntFloat$". Por conta dessa decisão, os visitors dos nós *Call* e *PexpFunc* que são nós de chamada de funções sofreram adaptações com o intuito reconstruir o padrão de id criado anteriormente.

Os operadores binários *Add*, *Mult*, *Diff*, *Div*, *Mod*, *Noeq* e *And* seguem os exemplos apresentados nas aulas. Outros nós como *If* , *If_else* , *Deny*, *Minus*, *Iterate* seguem os exemplos das aulas ou são implementações simples.

A função de *Read* considera a leitura apenas de variáveis do tipo Inteiro.

A verificação dos tipos de dados heterogêneos depende da estrutura *datas* mencionada anteriormente. Tanto na definição quanto na verificação desses tipos a classe *TyData* é utilizada. Entretanto apenas em *datas* estão guardados os ids e os tipos dos atributos definidos, sendo necessário uma consulta a *datas* ao efetuar uma verificação. Em caso de sucesso, um novo nó *TyData* é criado e inserido na pilha contendo apenas o id do tipo verificado.

3 Conclusão

Este relatório apresentou o processo de desenvolvimento de um analisador semântico para a linguagem lang. Foi escolhido o padrão de projeto Visitor, a ferramenta ANTLR fornece uma opção para atravessar á árvore de sintaxe, bastando apenas definir como cada nó deve ser visitado. O desenvolvimento deste projeto permitiu a continuidade do contato com o padrão de projeto Visitor, e também serviu como complemento do conteúdo exibido nas aulas.

A dificuldade na realização do trabalho foi a implementação dos métodos correspondentes a chamada de função, seja nos comandos ou nas expressões, já que tanto os parâmetros quanto os retornos devem ser bem tipados.