Teoria dos Compiladores - Analisador Semântico

UFJF

Matheus Gomes Luz Werneck (201835037)
Pedro Henrique Almeida Cardoso Reis (201835039)

Juiz de Fora, 25 de Junho de 2023

Contents

1 Introdução

Este relatório do trabalho de Teoria dos Compiladores tem como objetivo explicar primeiramente como foi implementado o Analisador Semãntico da linguagem *Lang*. O trabalho foi implementado em Java além de usar o ANTLR como uma ferramenta para nos auxiliar no desenvolvimento do código.

2 Instruções para o uso do projeto

No projeto encontram-se dois diferentes Shell Scripts para a compilação do mesmo.

- **run-project.sh:** Em resumo, este primeiro Shell Script remove o arquivo de log do código desenvolvido e logo depois compila as gramáticas desenvolvidas com o auxílio do *ANTLR* e, usando o *Maven*, executa o projeto.

2.1 Como rodar o projeto

- Certifique de ter instalado em seu computador o Java, JDK e Maven.
- Para executar o programa, abra um terminal,
  - Execute o comando **mvn clean install** para instalar as dependências do projeto.

    ```
    mvn clean install
    ```

  - Execute o comando **./run-project.sh (opção)**. Passando -bs é rodado todos os teste sintáticos, -byt para rodar os semânticos, ou passe o caminho de algum arquivo para passar pelo sintático, semântico e depois interpretar o mesmo;
    Pode ser necessário utilizar o comando *chmod +x run-project.sh* para dar permissão de execução ao script caso esteja usando Linux.

    ```
    ./run-project.sh some_file.txt
    ```

  - Após a execução do script, será impresso na tela a interpretação do arquivo *sample.txt*

3 Tomada de decisões de implementação

Antes de nos aprofundarmos no Analisador Léxico deixaremos registrado nessa seção algumas das decisões tomadas durante a implementação do projeto.

A primeira das decisões tomadas foi a utilização do Maven para fazer o Build do

projeto. Optamos pelo Maven pois segue um padrão de diretório e convenções predefinidas para estruturar projetos Java, uma vez que estávamos tendo problemas na hora de encontrar pacotes e classes para fazer a compilação. Graças ao Maven ficou mais fácil de entender e estruturar o projeto.

Outra decisão importante foi a utilização da biblioteca *Log4j*. Com a utilização de logs ficou mais fácil para entender os erros, além de poder visualizar o passo-a-passo de como o nosso código estava se comportando ao longo de seu desenvolvimento.

Para a implementação do analisador semântico foi utilizado o padrão Visitor, foi criado uma classe que implementa o Visitor chamada TypeCheckVisitor e essa classe faz toda a avaliação semântica do código, e no final printa os erros na tela se houverem.

4 Analisador Semântico

A terceira parte do trabalho de Teoria dos Compiladores é a entrega do Analisador Semântico. Esse trabalho, comparado ao anterior, possui uma pasta chamada *typeCheckUtils*. Nesta pasta encontram-se algumas classes para checagem de tipos, como por exemplo a classe *STyArr.java*, responsável por fazer a verificação do tipo Array.

```java
package com.compiler.typeCheckUtils;

public class STyArr extends SType {

    private SType a;

    public STyArr(SType t) {
        a = t;
    }

    public SType getArg() {
        return a;
    }

    public boolean match(SType v) {
        return (v instanceof STyErr) || (v instanceof STyArr) && (a.match(((STyArr) v).getArg()));
    }

    public String toString() {
        return a.toString() + "[]";
    }

}

```

Figure 1: Classe STyArr.java

Logo abaixo segue alguma das principais classes do trabalho:

- **App.java:** É a classe "Main" do nosso trabalho. Sua função é interpretar um programa usando a biblioteca ANTLR. Nesta classe, fazemos a importações necessárias para o funcionamento do *ANTLR, AST e Visitors*. É aqui também que recebemos um argumento via linha de comando que especifica o arquivo contendo o programa a ser interpretado.

- **lang.g4:** O arquivo *lang.g4* é um arquivo de gramática ANTLR que descreve a estrutura da linguagem *lang*. Essa classe define as regras de análise sintática e também como os diferentes elementos do programa devem ser interpretados. É nas regras da gramática que é definido a estrutura básica do programa, como por exemplo, os tipos de dados, estrutruas condicionais e laços de iteração. Desse modo, essa classe realiza a interpretação da linguagem *lang* visitando à árvore de análise sintática, sendo que cada nó da árvore é processado de acordo com a lógica definida no *InterpretVisitor*.

- **TypeCheckVisitor.java:** Classe responsável por percorrer a árvore sintática gerada pelo compilador e realizar a verificação de tipos em cada nó da árvore. Além disso, ela implementa a interface Visitor e contém métodos de visita para cada tipo de nó da árvore.

4.1 Estrutura de Dados Utilizadas

Type Stack (Pilha de Tipos): É uma pilha utilizada para rastrear os tipos das expressões durante a análise semântica. À medida que as expressões são avaliadas, os tipos resultantes são empilhados. Isso é importante pois assim conseguimos realizar: verificações de tipo, verificar a compatibilidade de tipos em operações binárias, atribuições, chamadas de função, etc.

Error Lists(Listas de Erros): São estruturas de dados usadas para coletar e armazenar erros encontrados durante a análise semântica. Quando um erro é detectado, adicionamos esse mesmo erro à lista de erros ArrayList<String>.

5 Sobre o código

Nessa seção iremos apresentar as principais ideias/decisões tomadas para a implementação do Analisador Semântico.

5.1 read

O código da figura abaixo define um método chamado visit que recebe um objeto do tipo Read como parâmetro. O trecho de código da figura abaixo lida com uma instrução de leitura de uma variável. Ele verifica se a variável está declarada, realiza algumas verificações de tipo e adiciona a variável à tabela de símbolos, associando-a ao tipo "int". Se o tipo da variável não for "int", uma mensagem de erro é gerada. Em resumo, tomamos como decisão fixar o valor de read sendo um inteiro

```
public void visit(Read read) {

    String attributionId = read.getLvalue().getId();

    if (checkIfHasVariableDeclared(attributionId)) {
       read.getLvalue().accept(this);

       SType currentVarType = typeStack.pop();

       if (currentVarType.match(typeInt)) {
          this.env.peek().put(attributionId, typeInt);
       } else {
          addErrorMessage(read, TypeCheckUtils.createTypeErrorRead(attributionId, currentVarType));
       }
    } else {
       this.env.peek().put(attributionId, typeInt);
    }

 }
```

Figure 2: read

5.2 Atribuição

A atribuição é foi uma das partes mais complexas do trabalho pois nela existe a possibilidade de fazermos a atribuição de um objeto, a atribuição de um Array, de um array de objetos, ou de um tipo básico, logo há varias possiblidades o que levou a uma dificuldade na implementação. O código abaixo verifica se os tipos das expressões de atribuição estão corretos, realiza validações específicas para atribuições a arrays e atribuições a atributos de objetos. Em caso de alguma inconsistência, mensagens de erros são geradas. Além disso, ele atualiza o .env de tipos com as novas atribuições corretas.

```
public void visit(Attribution attr) {
   LValue id = attr.getID();

   if (id instanceof ArrayPositionAccess) {
      id.accept(this);

      SType expectedType = typeStack.pop();

      attr.getExp().accept(this);

      SType exprType = typeStack.pop();

      String idWithIndex = id.getId() + "[]";

      if (!expectedType.match(exprType))
         addErrorMessage(id, TypeCheckUtils.createVariableRedeclarationMessage(expectedType, exprType, idWithIndex));

   }

   // x.z = 10
   else if (id instanceof AttributeAccess) {
      AttributeAccess access = (AttributeAccess) id;
      LValue leftSideId = access.getLeftValue();
      String rightSideId = access.getAcessId().getId();

      leftSideId.accept(this);

      SType varType = typeStack.pop();

      if (varType instanceof STyData) {
         STyData var = (STyData) varType;
         STyData data = datas.get(var.getId());

         if (data.getVars().containsKey(rightSideId)) {
            attr.getExp().accept(this);
            SType val = typeStack.pop();
            SType expectedType = data.getVars().get(rightSideId);

            if (val.match(data.getVars().get(rightSideId))) {

               if (!(leftSideId instanceof ArrayPositionAccess)) {
                  var.add(access.getAcessId().getName(), val);
                  env.peek().put(id.getId(), val);
               }
            } else {
               addErrorMessage(id,
                     TypeCheckUtils.createVariableRedeclarationMessage(expectedType, val, rightSideId));
            }

         } else {
            addErrorMessage(leftSideId,
                  TypeCheckUtils.createObjectInvalidAttributeMessage(rightSideId, leftSideId.getId()));
         }

      }

   } else {
      attr.getExp().accept(this);

      SType val = typeStack.pop();

      if (this.env.peek().containsKey(id.getId())) {
         id.accept(this);
         SType currentType = typeStack.pop();

         if (!currentType.match(val)) {
            addErrorMessage(id, TypeCheckUtils.createVariableRedeclarationMessage(currentType, val, id.getId()));
            val = typeErr;
         }
      }

      env.peek().put(id.getId(), val);
   }

}
```

Figure 3: atribuicao

5.3 Função

Na análise semântica da função, foram verificados os seguintes pontos: primeiro, se os parâmetros passados na chamada de função batem em tipo e a sua quantidade de parâmetros com os específicados na função; Também é verificado se os tipos de retorno batem com os tipos especificados nas variáveis de retorno; Abaixo segue como exemplo o código da função

```
public void visit(FunctionCall functionCall) {

    Function func = functions.get(functionCall.getFunctionName());

    if (func != null) {
       int receivedParams = functionCall.getParams().size();

       if (func.isQuantityOfParamsValid(receivedParams)) {
          for (Expr expr : functionCall.getParams())
             expr.accept(this);

          func.accept(this);

          if (functionCall.getReturnsId().size() > 0 && func.getReturns().size() == 0) {
             addErrorMessage(func, "Function " + func.getName() + " doesnt return any value");
          }

          else if (functionCall.getReturnsId().size() > 0 && returnMode) {

             for (LValue returnId : functionCall.getReturnsId()) {
                String returnVariableName = returnId.getId();

                SType receivedType = typeStack.pop();
                SType expectedType = paramStack.pop();

                if (expectedType.match(receivedType)) {
                   if (this.env.peek().containsKey(returnVariableName)) {
                      SType currentType = this.env.peek().get(returnVariableName);

                      if (!currentType.match(expectedType))
                         addErrorMessage(func, TypeCheckUtils.createVariableRedeclarationMessage(expectedType,
                               currentType, returnVariableName));

                   } else {
                      this.env.peek().put(returnVariableName, receivedType);
                   }

                } else {
                   addErrorMessage(func,
                         TypeCheckUtils.createWrongFunctionReturnMessage(expectedType, receivedType,
                               func.getName()));

                   this.env.peek().put(returnVariableName, typeErr);
                }

             }

             returnMode = false;
          }
       }

       else {
          addErrorMessage(func, "Function " + functionCall.getFunctionName() + " expected "
                + func.getParamlist().size() + " params");

          this.typeStack.push(typeErr);
       }

    } else {
       String errMessage = "Function: " + functionCall.getFunctionName() + " Not declared";
       addErrorMessage(func, errMessage);
    }

 }

```

Figure 4: Função

6 Dificuldades

Ao longo da implementação, as maiores dificuldades observadas foram nas implementação envolvendo Array e Data. Na atribuição por exemplo, pode ocorrer atribuição de um novo Data ou um novo Data dentro de uma posição do Array. Para fazer a implementação dessa checagem os vários casos específicos que

podem acontecer nesse fluxo tornou a implementação trabalhosa.

```
public void visit(NewArray newArray) {

    newArray.getType().accept(this);
    SType returnedType;

    returnedType = paramStack.pop();

    newArray.getExpr().accept(this);

    SType indexType = this.typeStack.pop();

    if (indexType.match(typeInt)) {
       this.typeStack.push(new STyArray(returnedType));
    } else {
       addErrorMessage(newArray, TypeCheckUtils.createInvalidArrayIndexTypeMessage(returnedType));
       typeStack.push(typeErr);
    }

}
```

Figure 5: Um exemplo do array

```

public void visit(NewData data) {

    if (data.getType() instanceof TypeCustom) {
       data.getType().accept(this);

       SType returnedType = paramStack.pop();
       typeStack.push(returnedType);

    } else {
       addErrorMessage(data, data.getTypeName() + " cannot be instantiate with New");
       typeStack.push(typeErr);
    }

}
```

Figure 6: Um exemplo do data