Yuri Kayser da Rosa 12201035
Alexandre Pereira Back 18100844

Implementação

Atividade A

O primeiro passo tomado para a realização deste trabalho foi implementar a construção da árvore sintática da linguagem utilizando como referência o capítulo 6 do livro “Como construir um compilador utilizando ferramentas Java”.

Como a gramática implementada foi, ao longo dos últimos trabalhos, diferenciando-se significativamente da gramática implementada no livro, foi necessário implementar algumas mudanças.

A primeira mudança foi no ClassBodyNode, que em nosso trabalho contém apenas um ListNode recebido pelo não terminal classBodyList() contendo as estruturas definidas dentro do corpo da classe. Cada nó desta lista contém um ClassBodyStatementNode que é criado no não terminal classbodystatement().

O nó ClassBodyStatementNode contém 5 atributos representando as estruturas que podem ser definidas no corpo da classe. Cada instância de ClassBodyStatementNode possui apenas 1 destes atributos com valor não nulo. Devido a extensão destas duas estruturas, optamos por não replicá-las neste documento, mas elas podem ser vistas nos arquivos Fun.jj e ClassBodyStatementNode.java.

Também foi necessário criar estruturas adicionais para comportar a utilização dos operadores lógicos XOR,OR,AND e NOT.

Foram criados os nós XorOrNode, AndNode e NotNode para representar estes operadores. As classes XorOrNode e AndNode seguem a mesma lógica de implementação utilizada pelo nó AddNode e o NotNode foi baseado na implementação do UnaryNode.

```
public class XorOrNode extends ExpreNode{
   public ExpreNode expr1;
   public ExpreNode expr2;

   public XorOrNode(Token t, ExpreNode e1, ExpreNode e2) {
      super(t);
      expr1 = e1;
      expr2 = e2;
   }
}
```

```
ExpreNode expression(RecoverySet g) throws ParseEOFException:
{
        ExpreNode e1 = null, e2 = null;
        Token t = null;
}
{
        try
        {
                e1 = andexpression()
                (
                  ( t=<XOR> | t=<OR>) e2=andexpression()
                  { e1 = new XorOrNode(t,e1,e2);}
                )*
                {return e1; }
        }
        catch (ParseException e) {
                consumeUntil(g, e, "expression"); // usa o RS da origem
                return e1;
        }
}

public class AndNode extends ExpreNode{
   public ExpreNode expr1;
   public ExpreNode expr2;

   public AndNode(Token t, ExpreNode e1, ExpreNode e2) {
      super(t);
      expr1 = e1;
      expr2 = e2;
   }
}

ExpreNode andexpression() throws ParseEOFException:
{
        ExpreNode e1 = null, e2 = null;
        Token t = null;
}
{
  e1=avlexpression()
  (
    ( t=< AND > ) e2=avlexpression()
        { e1 = new AndNode(t,e1,e2);}
  )*
        {return e1;         }
}
```

```
public class NotNode extends ExpreNode{

	public ExpreNode expre;

	public NotNode(Token not, ExpreNode expre) {
		super(not);
		this.expre = expre;

	}

}

ExpreNode unaryexpr() throws ParseEOFException:
{
	ExpreNode e;
	Token t = null;
	Token not = null;
}
{
	[not = < NOT >] [(t=< PLUS >|t=< MINUS >)] e=factor()
	{
	  if(not!=null) {
		if(t!=null) {
			return new NotNode(not, new UnaryNode(t,e));
		} else {
			return new NotNode(not,e);
		}
	  }else {
		return ( (t == null) ? e : new UnaryNode(t, e));
	  }
	}
}
```

Atividade B

Depois de criar toda a estrutura de classes necessária para a representação e construção da árvore sintática, foram realizados os procedimentos para a impressão e numeração dos nós da árvore gerada, segundo como referência o capítulo 7 do livro “Como construir um compilador utilizando ferramentas Java”.

Algumas mudanças foram necessárias, embora grande parte da classe PrintTree tenha se mantido intacta. Para comportar os operadores lógicos foi necessário adicionar os metodos de contagem e impressão específicos para cada um dos nós AndNode, NotNode e XorOrNode. Também foi necessário inserir verificações de tipo nos métodos numberExpreNode() e printExpreNode() no mesmo padrão dos demais nós que extendem a classe ExpreNode.

Outra mudança necessária foi na impressão da lista de parâmetros das declarações de métodos, uma vez que esta lista pode conter tanto nós do tipo VarDeclNode quanto AtribStatNode. Para tratar este caso, foram criados os métodos abaixo:

```
public void numberParamList(ListNode x){
        if (x == null) {
       return;
    }

    x.number = kk++;
    if(x.node instanceof VarDeclNode){
        numberVarDeclNode((VarDeclNode) x.node);
    }else{
        numberAtribNode((AtribStatNode)x.node);
    }
    numberParamList(x.next);
}

public void printParamList(ListNode x){
        if (x == null) {
       return;
    }

    System.out.println();
    System.out.print(x.number + ": ListNode (ParamsList) ===> " +
       x.node.number + " " +
       ((x.next == null) ? "null" : String.valueOf(x.next.number)));
    if(x.node instanceof VarDeclNode){
        printVarDeclNode((VarDeclNode) x.node);
    }else{
        printAtribNode((AtribStatNode)x.node);
    }
    printParamList(x.next);
}
```

Foi necessário também criar novos métodos para a impressão do nó ClassBodyStatementNode levando em conta o parâmetro não nulo de cada instancia deste nó que representa uma instrução inserida no corpo da classe. Os métodos criados seguem abaixo:

```
public void numberClassBodyStatementNode(ClassBodyStatementNode x){
    if (x == null) {
       return;
    }
    x.number = this.kk++;
    if(x.classList!=null){
        numberClassDeclListNode(x.classList);
    }else if(x.atributionStatement!=null){
        numberAtribNode(x.atributionStatement);
    }else if(x.construct!=null){
        numberConstructDeclNode(x.construct);
    }else if(x.method!=null){
        numberMethodDeclNode(x.method);
    }else{
```

```
            numberVarDeclNode(x.varDeclaration);
        }
    }

    public void printClassBodyStatementNode (ClassBodyStatementNode x){
        if (x == null) {
            return;
        }
        System.out.println();
            System.out.print(x.number + ": ClassBodyStatementNode ===> " + ((x.classList == null) ?
"null" : String.valueOf(x.classList.number)) + " "
        + ((x.atributionStatement == null) ? "null" : String.valueOf(x.atributionStatement.number)) + " "
        + ((x.construct == null) ? "null" : String.valueOf(x.construct.number)) + " "
        + ((x.method== null) ? "null" : String.valueOf(x.method.number)) + " "
        + ((x.varDeclaration== null) ? "null" : String.valueOf(x.varDeclaration.number)) + " "
        );

        if(x.classList!=null){
            printClassDeclListNode(x.classList);
        }else if(x.atributionStatement!=null){
            printAtribNode(x.atributionStatement);
        }else if(x.construct!=null){
            printConstructDeclNode(x.construct);
        }else if(x.method!=null){
            printMethodDeclNode(x.method);
        }else{
            printVarDeclNode(x.varDeclaration);
        }
    }
```

Após realizadas as alterações necessárias foram realizados testes para verificar a corretude da implementação. Não encontramos erros de representação da árvore sintática em nossos testes.

Atividade C

Para implementação da tabela de símbolos, seguiu-se o passo-a-passo do livro. Conforme requisitado pelo professor, também foi adicionado o atributo *optional* do tipo *boolean* na classe EntryRec, para definir se um parâmetro de método é opcional(TRUE) ou obrigatório(FALSE).

A princípio, não julgamos ser necessária nenhuma alteração a tabela de símbolos base do livro, acreditamos que no decorrer da implementação da análise semântica(atividade D), serão identificadas necessidades de melhorias que serão feitas e reportadas devidamente neste relatório.

Atividade D

De início, a atividade D aparentou ser bem tranquila para se implementar o passado no livro. Para atender ao pedido de exibir uma mensagem sempre que uma classe fosse

encontrada, foi adicionado um System.out.println() no método ClassCheckClassDeclNode da classe ClassCheck.

Além disto, devido a mudança da linguagem do livro para a proposta neste trabalho, como podemos ter classes instanciadas em diversas ordens dentro de um programa, na classe ClassCheck foi alterado método ClassCheckClassDeclListNode para poder chamar também o método ClassCheckClassBodyStatementNode, visto que um ClassBodyStatementNode pode possuir classes instanciadas dentro dele, chamando novamente o método ClassCheckClassDeclListNode, que por sua vez chamará o ClassCheckClassDeclNode, que aí sim adicionará a classe nova na tabela de símbolos.

Para averiguação de classes já existentes, também foi necessária uma pequena mudança no método classFindUp da classe Symtable. Aonde ele usava meramente um p.equals para comparação, foi alterado para p.name.equals, visto que não fora implementado um *override* do método equals da classe EntryTable.