Implementando um Analisador Sintático

ÉFREN L. SOUZA

Linguagem & Gramática

DL (*Didactic Language*)

```
programa maior_num inicio
   inteiro a;
   inteiro b;
   a = 2;
   b = 3;
   se (a > b) escreva(a);
   se (a <= b) escreva(b);
fim.
```

Gramática Reduzida

- Menos comandos
- Menos tipos
- Expressões incompletas
- **Porém possui toda a base para acrescentar os demais comandos e expressões**

```
PROGRAM     ::= programa ID BLOCK .
BLOCK       ::= inicio STMTS fim
STMTS       ::= STMT; STMTS | ε
STMT        ::= BLOCK | DECL | WRITE |
                        ASSIGN | IF
DECL        ::= TYPE ID
WRITE       ::= escreva(ID)
ASSIGN      ::= ID = EXPR
IF          ::= se (EXPR) STMT
EXPR        ::= EXPR "|" REL | REL
REL         ::= REL < ARITH | REL <= ARITH |
                        REL > ARITH | ARITH
ARITH       ::= ARITH + TERM | ARITH - TERM |
                        TERM
TERM        ::= TERM * FACTOR | FACTOR
FACTOR      ::= (EXPR) | ID | LIT_INT |
                        LIT_REAL | LIT_BOOL
```

Definições Regulares

- As definições regulares já estão estabelecidas no *lexer*

```
TYPE        ::= inteiro | real | booleano
ID          ::= LETTER(LETTER | DIGIT)*
LIT_INT     ::= DIGIT+
LIT_REAL    ::= DIGIT+.DIGIT*
LIT_BOOL    ::= verdadeiro | falso
LETTER      ::= a|b|...|z|A|B|...|Z|_
DIGIT       ::= 0|1|2|...|9
```

Análise Sintática

- Vamos implementar o *parser* em duas etapas:
  1. Verificação da sintaxe
  2. Produção da árvore sintática

Importando o Projeto para o Eclipse

1. Clique no menu `File -> Import`
2. Selecione a opção `General -> Existing Projects into Workspace`
3. Marque a opção `Select archive file`
4. Selecione o arquivo do projeto `dl_short_1.zip`
5. Clique em `Finish`

Organização do Projeto

- O projeto está organizado da seguinte forma

```
|--> dl
       |--> DL.java
|--> lexer
       |--> Lexer.java
       |--> Tag.java
       |--> Token.java
|--> prog.dl
```

- Por enquanto, o programa está apenas imprimindo os *tokens*

Verificando a Sintaxe

Classe `Parser`

- É a classe responsável pela análise sintática
- Na pasta `src`, crie um pacote chamado `parser`
- Na pasta `parser` crie uma classe chamada `Parser`

Parser.java

- Possui como atributo o *lexer* para obter os *tokens*
- Possui um *lookahead* para guardar o *token* atual
- O método `move()` serve para avançar para o próximo *token* sem verificá-lo

```
public class Parser {
    private Lexer lexer;
    private Token look;

    public Parser(Lexer lex) {
        lexer = lex;
        move();
    }

    private Token move()  {
        Token save = look;
        look = lexer.nextToken();
        return save;
    }
```

error() & match()

- O método `error()` lança erros sintáticos
- O método `match()` verifica o *token* atual antes de avançar

```
private void error(String s) {
    System.err.println("linha "
            + Lexer.line()
            + ": " + s);
    System.exit(0);
}

private Token match(Tag t) {
    if ( look.tag() == t )
        return move();
    error("Símbolo inesperado");
    return null;
}
```

parse()

- Método `parse()` inicia a análise sintática
- Ele chama o método que representa o não-terminal inicial da gramática

```
public void parse() {
    program();
}
```

program()

- Método program() verifica o cabeçalho e o corpo principal do programa

```
PROGRAM  ::= programa ID BLOCK .
```

```
private void program() {
        match(Tag.PROGRAM);
        match(Tag.ID);
        block();
        match(Tag.DOT);
        match(Tag.EOF);
}
```

block()

- Método `block()` gera uma lista de comandos
- O não-terminal $STMTS$ é suprimido do código
  - Removida recursão da calda
  - Métodos integrados

```
BLOCK    ::= inicio STMTS fim
STMTS    ::= STMT; STMTS | ε
```

```
private void block() {
    match(Tag.BEGIN);
    while( look.tag() != Tag.END) {
        stmt();
        match(Tag.SEMI);
    }
    match(Tag.END);
}
```

stmt()

- Método `stmt()` verifica todos os comandos da linguagem
- Por enquanto vamos considerar apenas os comandos que não dependem de expressões

STMT ::= BLOCK | DECL | ASSIGN | IF

```
private void stmt() {
    switch ( look.tag() ) {
    case BEGIN: block(); break;
    case INT: case REAL:
        case BOOL: decl(); break;
    default: error("comando inválido");
    }
}
```

decl()

- A primeira linha desse método não precisa fazer o `match()` porque isso já foi feito em `stmt()`

```
DECL      ::= TYPE ID
```

```
private void decl() {
    move();
    match(Tag.ID);
}
```

DL.java

- Antes de irmos para as expressões já podemos fazer um teste
- Faça testes com o programa fonte
- Verifique os erros sintáticos

```
public class DL {
    public static void main(String[] args) {
        Lexer l = new Lexer(
                new File("prog.dl"));
        Parser p = new Parser(l);
        p.parse();
        System.out.println("finalizado");
    }
}
```

stmt()

- Agora vamos para os comandos que dependem de expressões
- Vamos começar com as atribuições

STMT ::= BLOCK | DECL | ASSIGN | IF

```
private void stmt() {
    switch ( look.tag() ) {
    case BEGIN: block(); break;
    case INT: case REAL:
        case BOOL: decl(); break;
    case ID: assign(); break;
    default: error("comando inválido");
    }
}
```

assign()

```
ASSIGN    ::= ID = EXPR
```

```
private void assign() {
    match(Tag.ID);
    match(Tag.ASSIGN);
    expr();
}
```

expr()

- Remover a recursão à esquerda
- Depois reduzimos o código para chegar ao resultado ao lado

```
EXPR ::= EXPR "|" REL | REL
```

```
EXPR ::= REL EXPR'
EXPR'::= "|" REL EXPR' | ε
```

```
private void expr() {
    rel();
    while( look.tag() == Tag.OR ) {
        move();
        rel();
    }
}
```

rel()

- Como fica a gramática após remover a recursão à esquerda?
- Como fica o código simplificado?

```
REL  ::= REL < ARITH | REL <= ARITH |
         REL > ARITH | ARITH
```

```
REL  ::= ARITH REL'
REL' ::= < ARITH REL' | <= ARITH REL' |
         > ARITH REL' | ε
```

```
private void rel() {
    arith();
    while ( look.tag() == Tag.LT ||
            look.tag() == Tag.LE ||
            look.tag() == Tag.GT) {
        move();
        arith();
    }
}
```

arith()

- Como fica a gramática após remover a recursão à esquerda?
- Como fica o código simplificado?

```
ARITH ::= ARITH + TERM | ARITH – TERM |
          TERM
```

```
ARITH  ::= TERM ARITH’
ARITH’ ::= +TERM ARITH’ | -TERM ARITH’ |
           ε
```

```
private void arith() {
    term();
    while(  look.tag() == Tag.SUM ||
            look.tag() == Tag.SUB ) {
        move();
        term();
    }
}
```

term()

- Como fica a gramática após remover a recursão à esquerda?
- Como fica o código simplificado?

```
TERM  ::= TERM * FACTOR | FACTOR
```

```
TERM  ::= FACTOR TERM'
TERM' ::= * FACTOR TERM' | ε
```

```
private void term() {
    factor();
    while(  look.tag() == Tag.MUL ) {
        move();
        factor();
    }
}
```

factor()

- **Agora o código pode ser testado. Verifique a sintaxe das expressões com atribuições!!!**

```
FACTOR ::= (EXPR) | ID | LIT_INT |
                    LIT_REAL | LIT_BOOL
```

```
private void factor() {
    switch( look.tag() ) {
    case LPAREN: move(); expr();
        match(Tag.RPAREN); break;
    case LIT_INT: move(); break;
    case LIT_REAL: move(); break;
    case TRUE: case FALSE:
        move(); break;
    case ID: match(Tag.ID); break;
    default:
        error("expressão inválida");
    }
}
```

ifStmt()

- Como fica o código?
- Não esqueça de incluí-lo como comando
- **Teste o código. Verifique a sintaxe do comando se!!!**

```
IF          ::= se (EXPR) STMT
```

```
private void ifStmt() {
        match(Tag.IF);
        match(Tag.LPAREN);
        expr();
        match(Tag.RPAREN);
        stmt();
}
```

writeStmt()

- Como fica o código?
- Não esqueça de incluí-lo como comando
- **Teste o código. Verifique a sintaxe do comando write!!!**

```
WRITE   ::= escreva(ID)
```

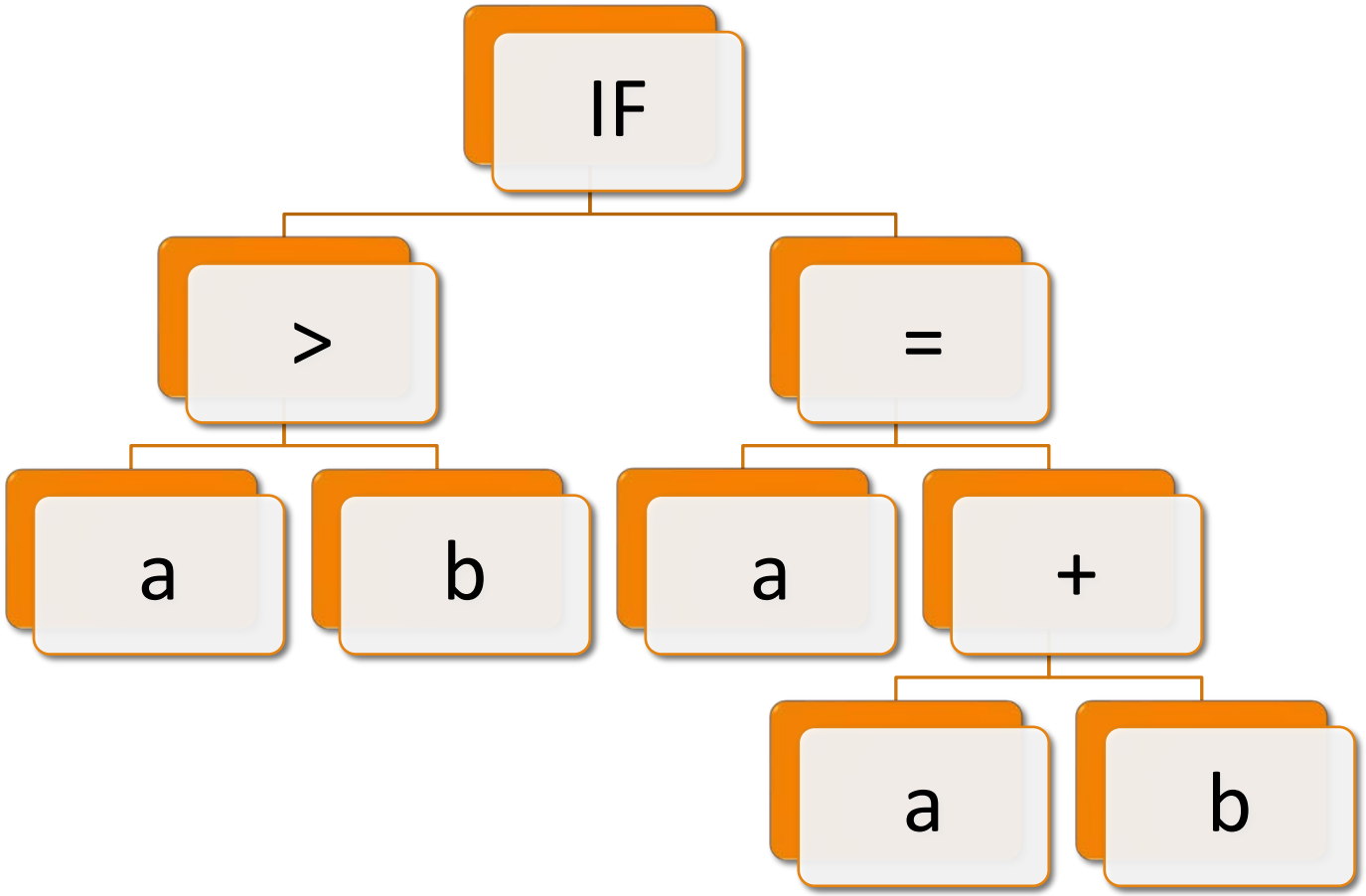
```
private void writeStmt() {
    move();
    match(Tag.LPAREN);
    match(Tag.ID);
    match(Tag.RPAREN);
}
```

Árvore Sintática

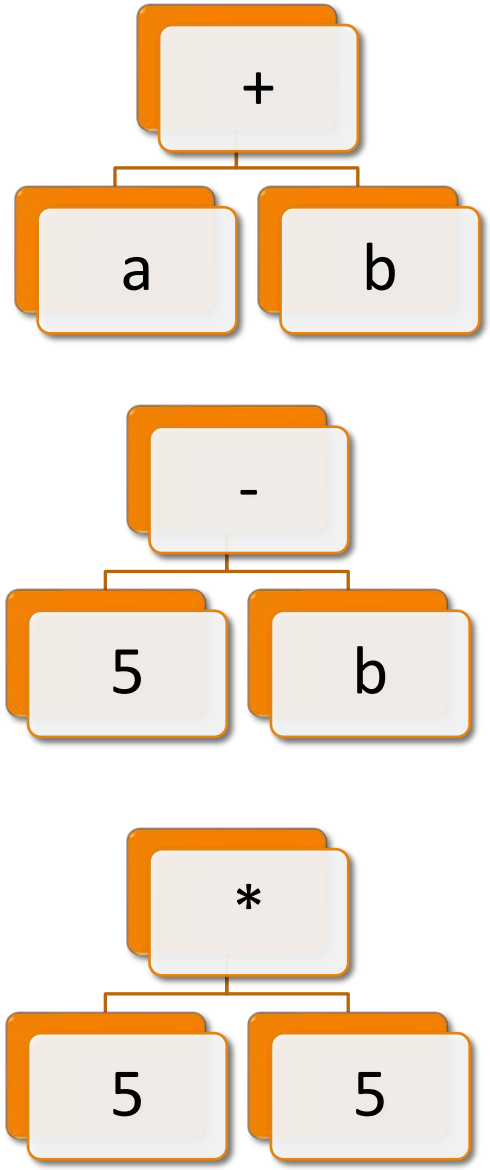
Árvore Sintática

- Precisamos criar uma árvore como esta
- Para isso, precisamos criar nós para comandos e expressões
- Cada nó tem um conjunto de nós filhos

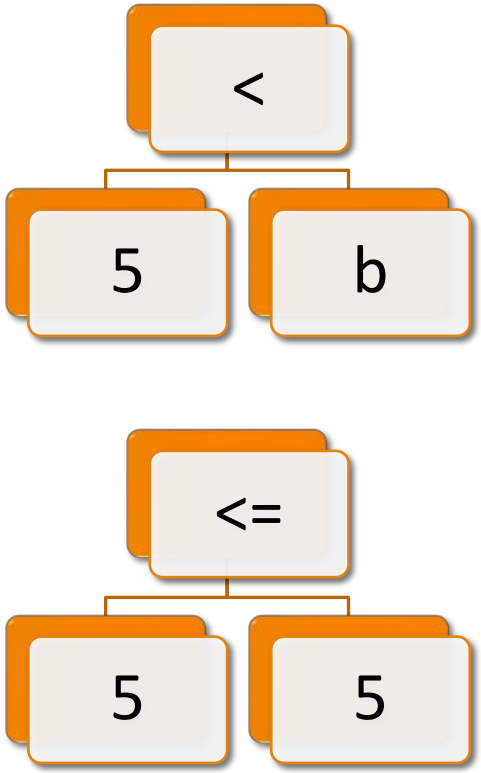
```
se  (a > b) a = a + b;
```



Binário



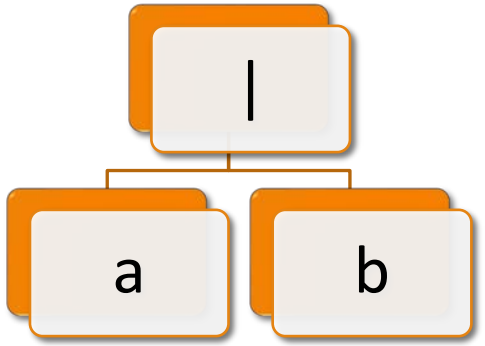
Relacionais



ID

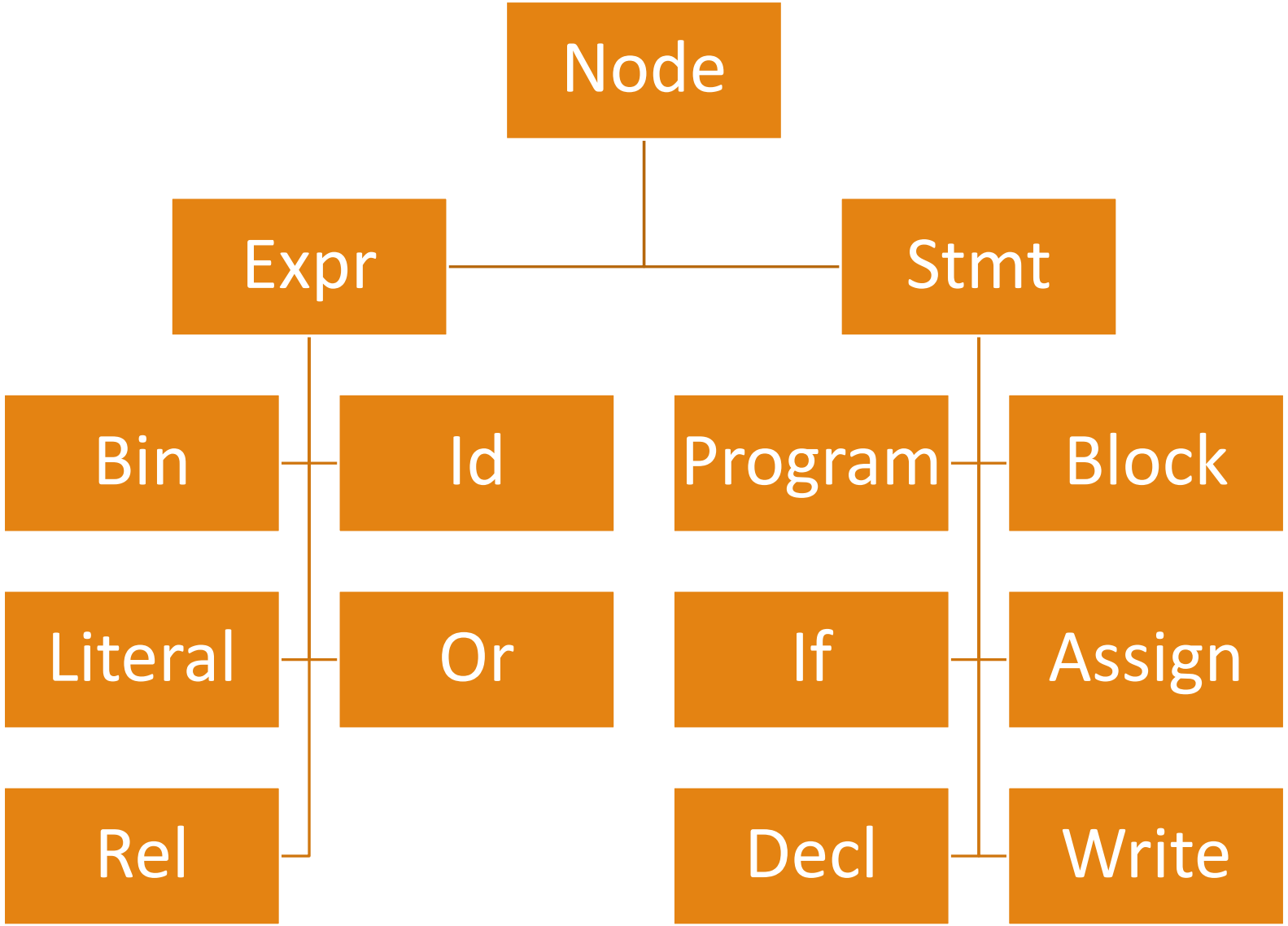
Literal

Ou



Hierarquia dos Nós

- Os nós serão representados por classes
- Tudo no programa é um **comando** ou uma **expressão**



Node.java

- Em `src` crie o pacote `inter` e dentro dele a classe `Node`
- É uma classe abstrata do qual todos os nós deverão derivar
- Possui uma lista que conterá todos os filhos de um determinado nó
- Os nós são usados posteriormente na análise semântica e geração de código

```
public abstract class Node {
    private LinkedList<Node> children =
            new LinkedList<Node>();

    protected void addChild( Node n ) {
        children.add(n);
    }

    protected LinkedList<Node> children() {
        return children;
    }
}
```

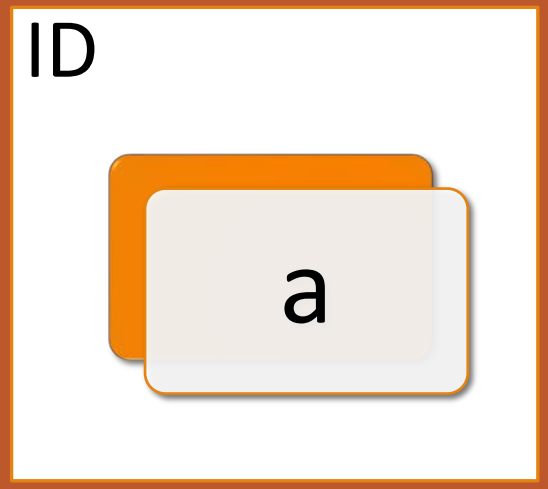
Expr.java

- crie o pacote `inter.expr` e dentro dele a classe `Expr`
- Também é uma classe abstrata do qual todas as expressões vão derivar
- Toda expressão tem pelo menos um operador/operando de um determinado tipo

```
public abstract class Expr extends Node {
    protected Token op;
    protected Tag type;

    public Expr(Token op, Tag type) {
        this.op = op;
        this.type = type;
    }

    public Token op() { return op; }

    public Tag type() { return type; }

    @Override
    public String toString() {
        return op.tag().toString();
    }
}
```

Id.java

- No pacote `inter.expr` crie a classe `Id`
- Esse nó não possui filhos




```java
public class Id extends Expr {
    public Id(Token op, Tag type) {
        super(op, type);
    }

    @Override
    public String toString() {
        return "%" + op.lexeme();
    }
}
```

Literal.java

- No pacote `inter.expr` crie a classe `Literal`
- Também não possui filhos


```
public class Literal extends Expr {

    public Literal(Token op, Tag type) {
        super(op, type);
    }

    @Override
    public String toString() {
        switch (op.tag()) {
        case TRUE:
            return "true";
        case FALSE:
            return "false";
        default:
            return op.lexeme();
        }
    }
}
```

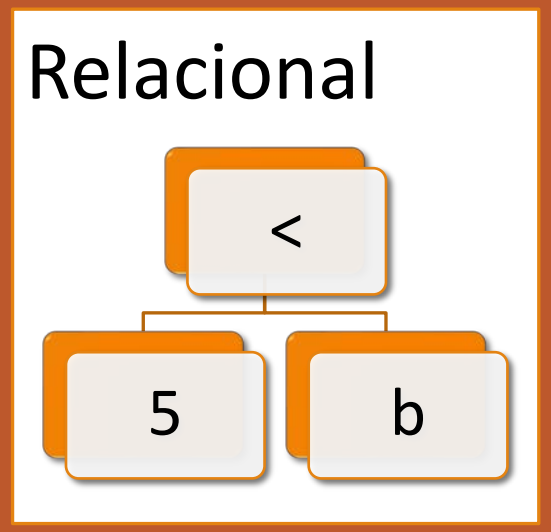
Bin.java

- No pacote `inter.expr` crie classe `Bin`
- Representa as operações aritméticas binárias
- O tipo dessa expressão vai depender do tipo de seus operandos


```
public class Bin extends Expr {
    protected Expr expr1;
    protected Expr expr2;

    public Bin( Token op, Expr e1, Expr e2 ) {
        super(op, null);
        expr1 = e1;
        expr2 = e2;
        addChild(expr1);
        addChild(expr2);
    }
}
```

Rel.java

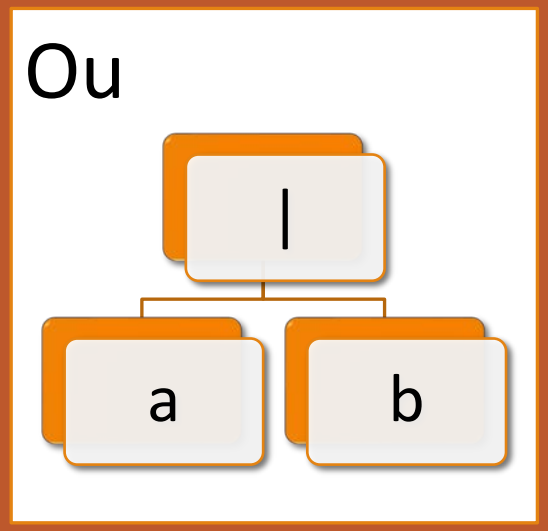
- No pacote `inter.expr` crie classe `Rel`
- Representa as operações relacionais
- O tipo dessa expressão é sempre booleano




```
public class Rel extends Expr {
    protected Expr expr1;
    protected Expr expr2;

    public Rel(Token op, Expr e1, Expr e2 ) {
        super(op, Tag.BOOL);
        expr1 = e1;
        expr2 = e2;
        addChild(expr1);
        addChild(expr2);
    }
}
```

Or.java

- No pacote `inter.expr` crie classe `Or`
- Representa o operador lógico Ou
- O tipo dessa expressão é sempre booleano




```
public class Or extends Expr {
    protected Expr expr1;
    protected Expr expr2;

    public Or(Expr e1, Expr e2) {
        super(new Token(Tag.OR, "|"),
                Tag.BOOL);
        expr1 = e1;
        expr2 = e2;
        addChild(expr1);
        addChild(expr2);
    }
}
```

Stmt.java

- Crie o pacote `inter.stmt` e dentro dele a classe `Stmt`
- Representa todos os comandos
- É uma classe abstrada

```
public abstract class
        Stmt extends Node {
}
```

Block.java

- Em `inter.stmt` crie a classe `Block`
- O bloco é uma lista de comandos
- Todo comando em um block é um de seus filhos

```
public class Block extends Stmt {

    public Block() {
    }

    public void addStmt(Stmt stmt) {
        addChild(stmt);
    }

    @Override
    public String toString() {
        return "BLOCK";
    }
}
```

Decl.java

- Em `inter.stmt` crie a classe `Decl`
- Declaramos uma variável por vez

```
public class Decl extends Stmt {
    private Id id;

    public Decl(Id i) {
        id = i;
        addChild(id);
    }

    @Override
    public String toString() {
        return "DECL";
    }
}
```

Assign.java

- Em `inter.stmt` crie a classe `Assign`
- Toda atribuição tem uma variável do lado esquerdo e uma expressão do lado direito

```
public class Assign extends Stmt {
    protected Id id;
    protected Expr expr;

    public Assign(Id i, Expr e) {
        id = i;
        expr = e;
        addChild(id);
        addChild(expr);
    }

    @Override
    public String toString() {
        return Tag.ASSIGN.toString();
    }
}
```

If.java

- Em `inter.stmt` crie a classe `If`
- Todo `if` tem uma expressão a ser avaliada e um corpo a ser executado

```
public class If extends Stmt {
    private Expr expr;
    private Stmt stmt;

    public If(Expr e, Stmt s) {
        expr = e;
        stmt = s;
        addChild(expr);
        addChild(stmt);
    }

    @Override
    public String toString() {
        return Tag.IF.toString();
    }
}
```

Write.java

- Em `inter.stmt` crie a classe `Write`
- Todo `Write` tem um Id que é a variável cujo valor será impresso

```
public class Write extends Stmt {
    private Id id;

    public Write(Id i) {
        id = i;
        addChild(id);
    }

    @Override
    public String toString() {
        return Tag.WRITE.toString();
    }
}
```

Program.java

- Em `inter.stmt` crie a classe `Program`
- Esse nó sempre será a raiz da árvore sintática

```
public class Program extends Stmt {
    private Token id;
    private Block block;

    public Program(Token i, Block b) {
        id = i; block = b;
        addChild(b);
    }

    @Override
    public String toString() {
        return Tag.PROGRAM.toString();
    }
}
```

Árvore no *parser*

- Até o momento, nosso parser apenas verificou a sintaxe

- Agora ele precisa criar um nó para cada comando e expressão

Parser.java

- Precisamos de um novo atributo para ser a raiz da árvore sintática
- No método `parse()` temos que definir a raiz da árvore sintática

```
public class Parser {
    private Lexer lexer;
    private Token look;
    private Node root;
```

```
    public void parse() {
        root = program();
    }
```

program()

- Para um nó Program, precisamos do seu nome e do seu bloco

```
private Program program() {
      match(Tag.PROGRAM);
      Token tokId = match(Tag.ID);
      Stmt b = block();
      match(Tag.DOT);
      match(Tag.EOF);
      return new Program(tokId, (Block)b);
}
```

block()

- Para um nó Block, precisamos adicionar os comandos, um de cada vez

```
private Stmt block() {
    Block b = new Block();
    match(Tag.BEGIN);
    while( look.tag() != Tag.END) {
        b.addStmt(stmt());
        match(Tag.SEMI);
    }
    match(Tag.END);
    return b;
}
```

stmt()

- O stmt() agora retorna os nós para cada comando

```
private Stmt stmt() {
        switch ( look.tag() ) {
        case BEGIN: return block();
        case INT: case REAL:
                case BOOL: return decl();
        case WRITE: return writeStmt();
        case ID: return assign();
        case IF: return ifStmt();
        default: error("comando inválido");
        }
        return null;
}
```

decl()

- O nó `Decl` precisa de um nó Id
- O nó `Id` precisa de um *token* para o id e de um *token* para o tipo

```
private Stmt decl() {
    Token type = move();
    Token tokId = match(Tag.ID);
    Id id = new Id(tokId, type.tag());
    return new Decl(id);
}
```

assign()

- O nó Assign precisa de um nó Id e de uma expressão

```
private Stmt assign() {
    Token tok = match(Tag.ID);
    Id id = new Id(tok, null);
    match(Tag.ASSIGN);
    Expr e = expr();
    return new Assign(id, e);
}
```

ifStmt()

- O nó If precisa de uma expressão e um comando

```
private Stmt ifStmt() {
        match(Tag.IF);
        match(Tag.LPAREN);
        Expr e = expr();
        match(Tag.RPAREN);
        Stmt s1 = stmt();
        return new If(e, s1);
}
```

writeStmt()

- O nó Write precisa de um Id

```
private Stmt writeStmt() {
        move();
        match(Tag.LPAREN);
        Token tok = match(Tag.ID);
        Id id = new Id(tok, null);
        match(Tag.RPAREN);
        return new Write(id);
}
```

expr()

- O nó Or precisa de duas expressões

```
private Expr expr() {
    Expr e = rel();
    while( look.tag() == Tag.OR ) {
        move();
        e = new Or(e, rel());
    }
    return e;
}
```

rel()

- O nó Or precisa de duas expressões

```
private Expr rel() {
    Expr e = arith();
    while ( look.tag() == Tag.LT ||
            look.tag() == Tag.LE ||
            look.tag() == Tag.GT) {
        Token op = move();
        e = new Rel(op, e, arith());
    }
    return e;
}
```

arith()

- O nó Bin precisa do operador e de duas expressões

```
private Expr arith() {
    Expr e = term();
    while(  look.tag() == Tag.SUM ||
            look.tag() == Tag.SUB ) {
        Token op = move();
        e = new Bin(op, e, term());
    }
    return e;
}
```

term()

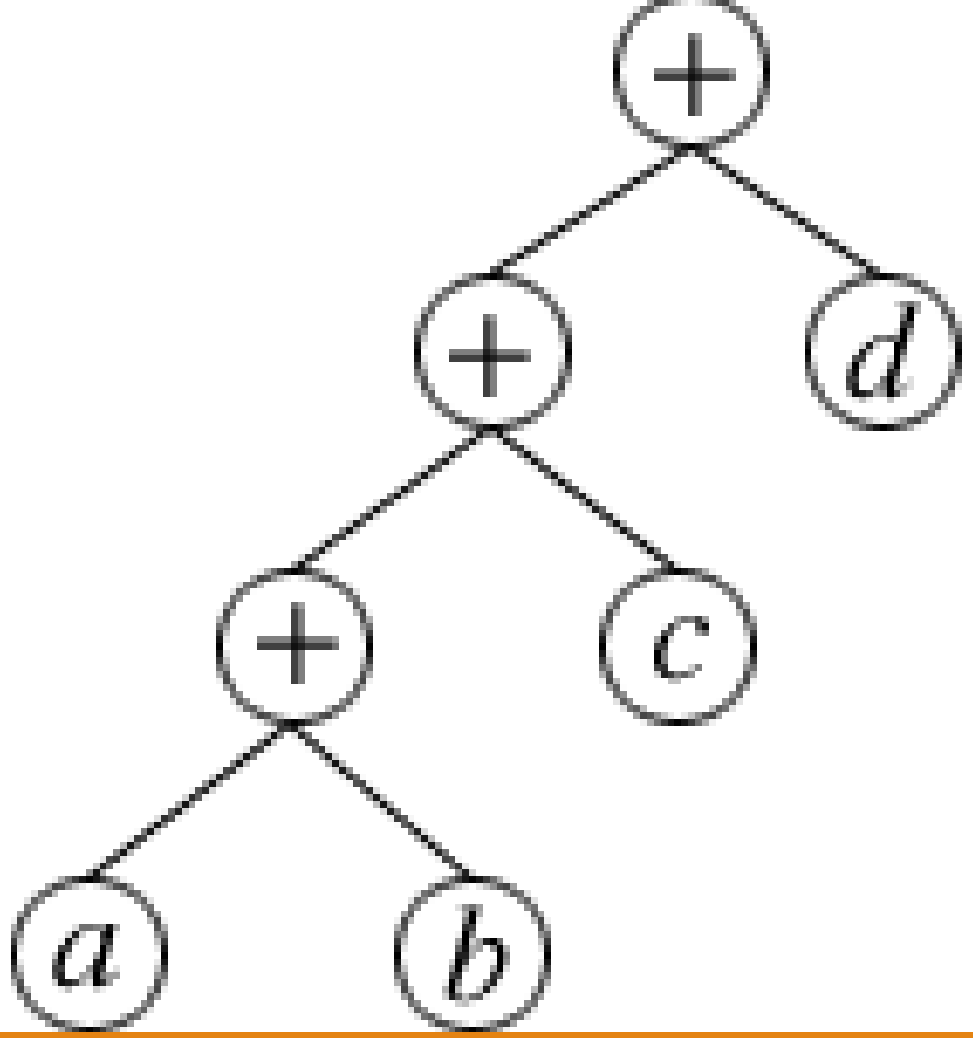
- O nó Bin precisa de um operador e duas expressões

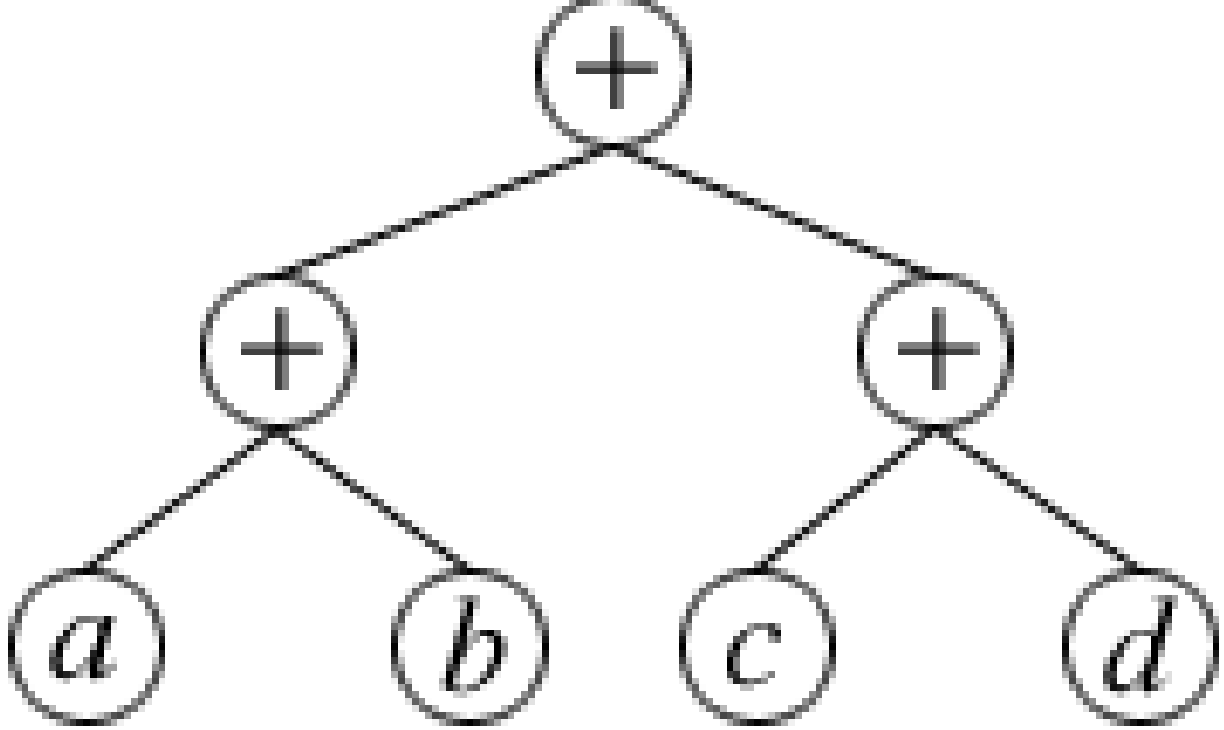
```
private Expr term() {
    Expr e = factor();
    while(  look.tag() == Tag.MUL ) {
        Token op = move();
        e = new Bin(op, e, factor());
    }
    return e;
}
```

factor()

- Os nós `Literal` e `Id` precisam de um operando e um tipo
- Por enquanto o tipo do ID fica nulo, pois não temos como determina-lo agora

```
private Expr factor() {
    Expr e = null;
    switch( look.tag() ) {
    case LPAREN: move(); e = expr();
        match(Tag.RPAREN); break;
    case LIT_INT:
        e = new Literal(move(), Tag.INT);
        break;
    case LIT_REAL:
        e = new Literal(move(), Tag.REAL);
        break;
    case TRUE: case FALSE:
        e = new Literal(move(), Tag.BOOL);
        break;
    case ID:
        Token tok = match(Tag.ID);
        e = new Id(tok, null); break;
    default: error("expressão inválida");
    }
    return e;
}
```





Apresentando a Árvore Sintática

Raiz

- Já temos a árvore sintática e acesso à sua raiz
- Agora temos que percorrer toda a árvore para imprimir seus nós

Node.java

```
public String strTree() {
    return strTree("");
}

private String strTree(String ident) {
    StringBuffer sb = new StringBuffer();
    sb.append(toString());
    for( Node n: children() ) {
        sb.append("\n" + ident + "|--> ");
        sb.append(n.strTree(ident + "      "));
    }
    return sb.toString();
}
```

Parser.java

```
public String parseTree() {
    return root.strTree();
}
```

DL.java

- Depois disso observe a árvore sintática
- Verifique a precedências das operações nas expressões

```
public static void main(String[] args) {
    //Análise
    Lexer l = new Lexer(
            new File("prog.dl"));
    Parser p = new Parser(l);
    p.parse();

    //Imprimindo a árvore sintática
    System.out.println(p.parseTree());
    System.out.println("finalizado");
}
```

Exercício

- Acrescente o comando while ao compilador de acordo com a produção ao lado

```
WHILE      ::= enquanto (EXPR) STMT
```

Bibliografia

- AHO, A. V.; SETHI, R.; ULLMAN, J. D. Compiladores: princípios, técnicas e ferramentas. Rio de Janeiro: LTC, 1995.
- CAMPBELL, B.; LYER, S.; AKBAL-DELIBAS, B. Introduction to Compiler Construction in a Java World. CRC Press, 2013.
- APPEL, A. W. Modern compiler implementation in C. Cambridge. Cambridge University Press, 1998.