Análise Semântica

ÉFREN L. SOUZA

Análise (*front-end*)

- Análise Léxica (*lexer*)
  - Detecta entradas com lexemas ilegais
- Análise Sintática (*parser*)
  - Detecta entradas com sentenças mal formadas
- Análise Semântica
  - Última fase da análise
  - Captura os erros restantes

Por que Análise Semântica?

- Há alguns erros nas linguagens de programação que a análise sintática não consegue capturar

- As gramáticas livres de contexto não são expressivas o suficiente para descrever tudo o que interessa na definição de uma linguagem

Papel da Análise Semântica

- Verificar a declaração de variáveis
- Verificar o tipos
- Converter tipos

Tabela de Símbolos e Escopo

Escopo

- O escopo é uma parte do programa à qual uma declaração se aplica
- Normalmente um bloco
- Dado o código C++ ao lado
  - Qual o escopo das variáveis?
  - Qual a saída gerada?

```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

B_1

B_2

B_3

B_4

Declaração	Escopo
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4`	B_4

Tabela de Símbolos

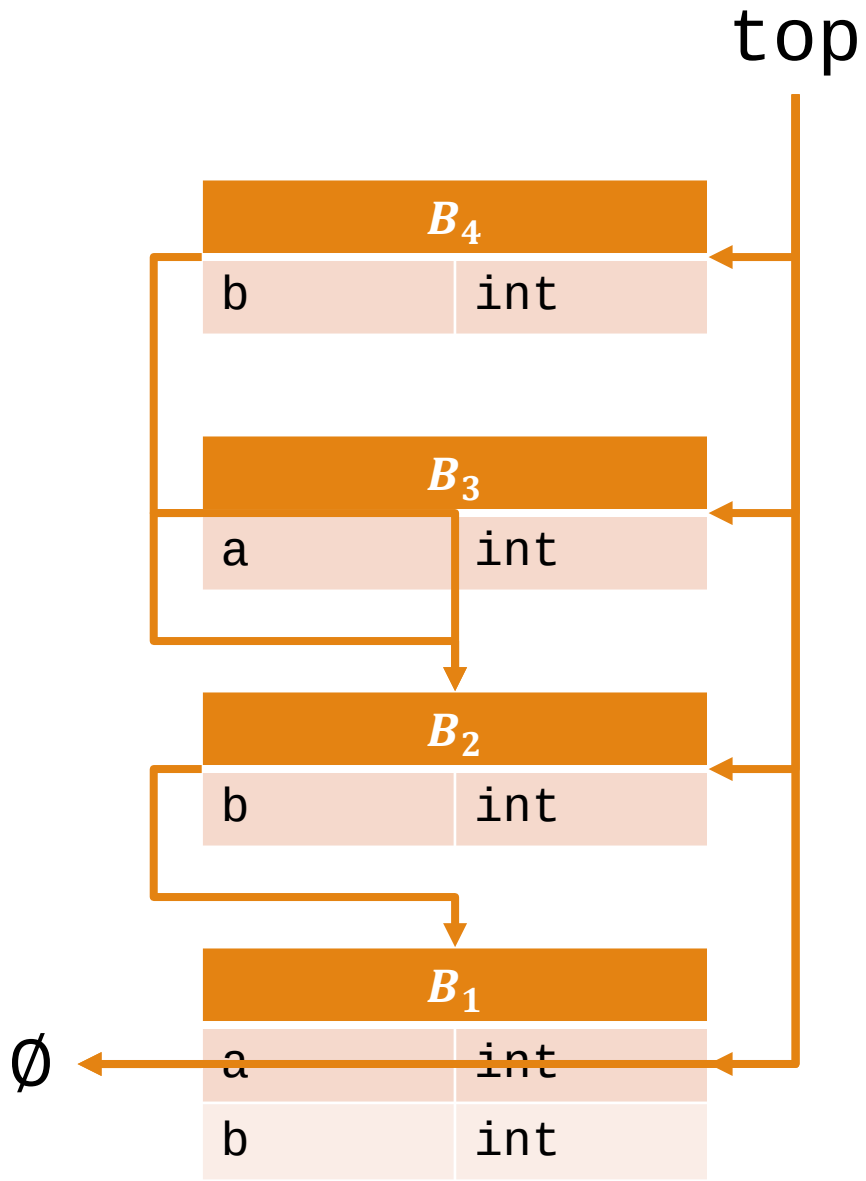
- É uma estrutura de dados usada para manter informações sobre os identificados
  - Nome, lexema, tipo, tamanho, endereço de memória...

- Essas informações são coletadas e usadas durante a análise
  - Criadas e recuperadas por ações semânticas

Tabela de Símbolos e Escopo

- Para implementar o escopo, cada bloco recebe sua própria tabela de símbolos
- Sempre que há uma declaração em um bloco, uma entrada é criada para essa variável na tabela desse bloco
- As tabelas de símbolos são encadeadas de acordo com o surgimento e fechamento dos blocos

```
main() {
    int a = 1;
    int b = 1;
    {
        int b = 2;
        {
            int a = 3;
            cout << a << b;
        }
        {
            int b = 4;
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```



Esquema de Tradução

- Como a tabela de símbolos é usada para:
  - Escopo
  - Declaração
  - Uso

```
PROGRAM ::= programa ID BLOCK .  { top = null;                }

BLOCK   ::= inicio               {      saved = top;
                                        top = new Env(top);   }
            STMTS
            fim                  {      top = saved;           }

DECL    ::= TYPE ID              {      s = new Symbol(ID);
                                        s.type = TYPE;
                                        top.put(ID.lexeme, s); }

FACTOR ::= ID                    {      s = top.get(id.lexeme); }
```

Checando Identificadores na DL

Escopo único

- Vamos implementar uma versão simplificada
- Vamos checar se os identificadores foram declarados
- Mas haverá apenas o escopo do bloco principal

Impontando 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_2.zip`
5. Clique em `Finish`

Estágio do Projeto

- Este projeto está:
  - Gerando expressões lógicas e aritméticas
  - Reconhecendo tipos e literais inteiros, reais e booleanos
  - Verificando toda a sintaxe e imprimindo a árvore sintática

```
programa teste inicio
   inteiro a; real b; booleano c;
   a = 2;
   b = 3.1415;
   c = falso;
   c = a < b;
   a = (a+2)*c;
   se ( c  verdadeiro )
        escreva(a);
fim.
```

Parser.java

- Temos que criar um atributo no parser que irá referenciar a tabela de símbolos

- Será usado um *hash*
  - A chave é o lexema do identificador
  - O valor é o nó desse identificador
  - temos consulta no tempo $O(1)$

```
public class Parser {
    private Lexer lexer;
    private Token look;
    private Node root;
    private Hashtable<String, Id> table;

    public Parser(Lexer lex) {
        lexer = lex;
        table = new Hashtable<String, Id>();
        move();
    }
```

decl()

- Antes de declarar uma nova variável, temos que verificar se ela não foi declarada antes
- Fazemos isso consultando a tabela de símbolos
  - Se o símbolo não existe na tabela, ele é adicionado
  - Se ele já existe, então ocorre um erro semântico
- **Teste o código tentando declarar uma mesma variável duas vezes!!!**

```
private Stmt decl() {
    Token type = move();
    Token tokId = match(Tag.ID);
    if (table.get(tokId.lexeme())==null){
        Id id = new Id(tokId, type.tag());
        table.put(tokId.lexeme(), id);
        return new Decl(id);
    }
    error("a variável '"
            + tokId.lexeme()
            + "' já foi declarada");
    return null;
}
```

assign()

- Antes de fazer uma atribuição, precisamos garantir que a variável já foi declarada
- Isso é feito consultado a tabela de símbolos
- O mesmo procedimento será feito sempre que o código fonte usar uma variável
- Então vamos criar um método para consultar a tabela

```
private Id findId( Token tokId ) {
    Id id = table.get(tokId.lexeme());
    if ( id == null )
        error("a variável '"
            + tokId.lexeme()
            + "' não foi declarada");
    return id;
}
```

assign()

- Se a variável não está na tabela de símbolos, ocorre um erro
- **Teste o código atribuindo um valor a uma variável que não foi declarada!**

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

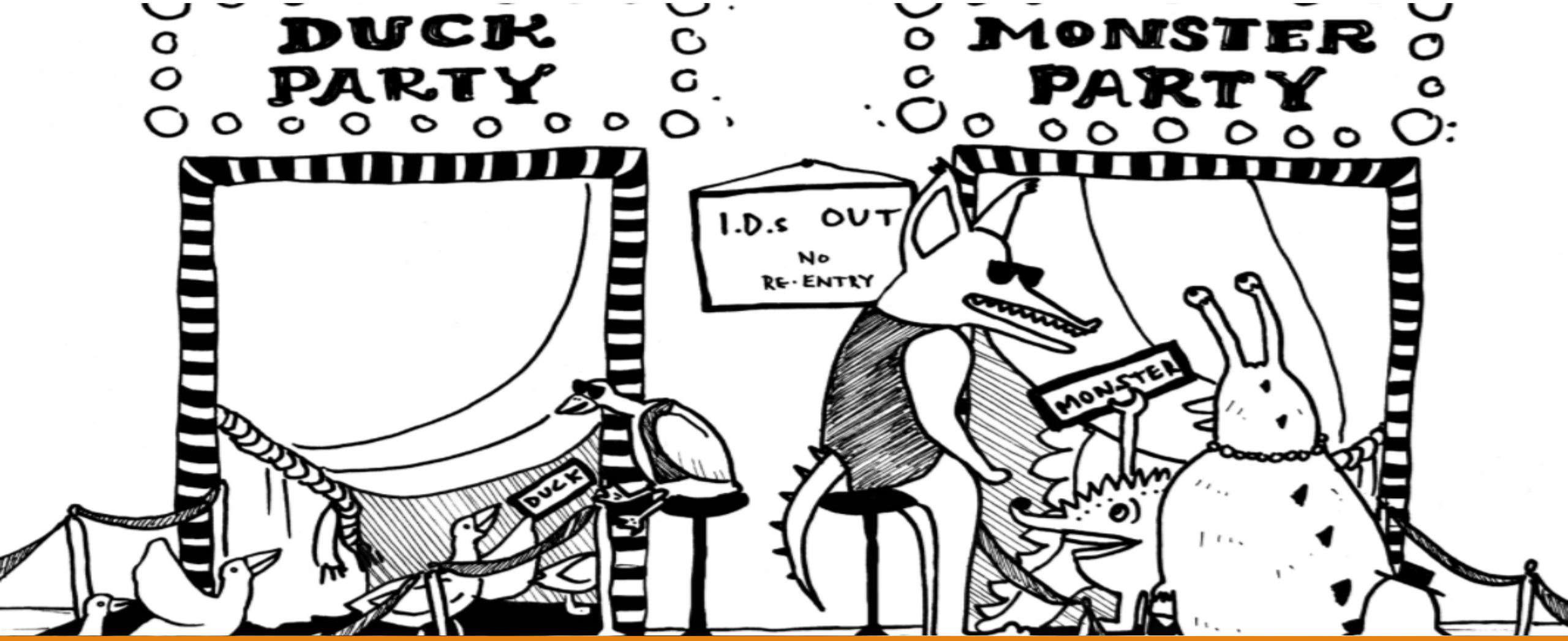
factor ()

- Quando uma variável é usada em uma expressão, também precisamos garantir que a variável já foi declarada
- **Teste o código usando uma variável não declarada em uma expressão!**

```
case ID:
    e = findId( match(Tag.ID) );
    break;
```

Exercício

Verifique a declaração de variável no comando `write`.



Verificação de Tipos

Verificação Estática & Dinâmica

- Verificação Estática
  - Analisa o programa em tempo de compilação
  - Nunca deixa os erros acontecerem em tempo de execução
- Verificação dinâmica
  - Checa as operações em tempo de execução
  - Facilita a prototipação
  - Mais flexível, porém menos eficiente

Sistema de Tipos

- Sistema fortemente tipado
  - Mais Robusto
  - Não permite qualquer tipo de erro
  - Java, Python, C#
- Sistema fracamente tipado
  - Permite alguns erros em tempo de execução
  - São mais rápidos
  - C/C++, PHP, JavaScript

Verificação de Tipo

- Garante que o tipo de uma construção combine com o tipo esperado

`if (expr) stmt`
- Espera-se que `expr` tenha o tipo boolean

`(a >= b)`
- Espera-se que `a` e `b` sejam numéricos - O resultado é do tipo booleano

Coersão

- Acontece quando o tipo de um operando é convertido pelo compilador para o tipo esperado
- A máquina só opera operandos de mesmo tipo

$$2 * 3.1415$$

- Usualmente o inteiro 2 é convertido para real 2.0
- Depois a multiplicação de ponto flutuante é realizada
- O resultado é um valor real

Verificando os Tipos na DL

Sistema de Tipos da DL

- É fortemente tipada
- A verificação é estática
- Verificamos os tipos no momento da criação dos nós

Tag.java

- Métodos auxiliares
- Identificam os tipos

```
public boolean isInt() {
    return this == Tag.INT;
}

public boolean isReal() {
    return this == Tag.REAL;
}

public boolean isBool() {
    return this == Tag.BOOL;
}
```

Tag.java

- Métodos auxiliares
- Identificam os tipos

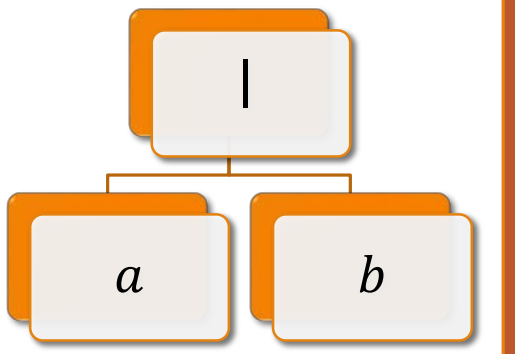
```
public boolean isNum() {
    return (isInt() || isReal());
}

public boolean isType() {
    return isNum() || isBool();
}
```

Node.java

- O método `error()` da classe Node serve para lançar erros semânticos

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

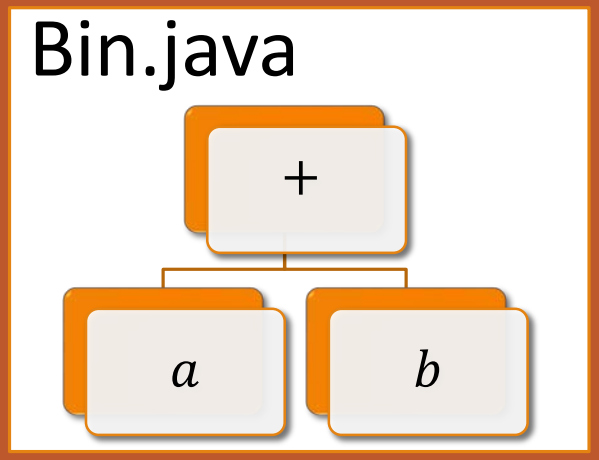
Or.java



- O operador lógico Ou só pode ser aplicado a operadores booleanos
- Então devemos checar os dois operandos
- **Teste o operador Ou!**

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

    public Or(Expr e1, Expr e2) {
        super(new Token(Tag.OR, "|"), Tag.BOOL);
        if ( !e1.type().isBool() ||
             !e2.type().isBool() )
            error("O operador lógico | só "
                  + "pode ser aplicado entre "
                  + "tipos booleanos");
        expr1 = e1;
        expr2 = e2;
        addChild(expr1);
        addChild(expr2);
    }
}
```

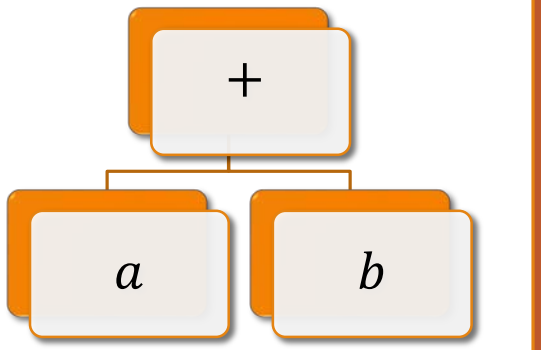
Bin.java



- Nossa linguagem deve permitir operações aritméticas entre reais e inteiros
- Sabemos que a máquina só pode operar dados de mesmo tipo
- Então temos que definir de que tipo será o operador
- O método `maxType()` retorna o tipo numérico que usa mais bits

```
private static Tag maxType(Tag t1, Tag t2) {
    if ( !t1.isNum() || !t2.isNum() )
        return null;
    else if ( t1.isReal() || t2.isReal() )
        return Tag.REAL;
    else
        return Tag.INT;
}
```

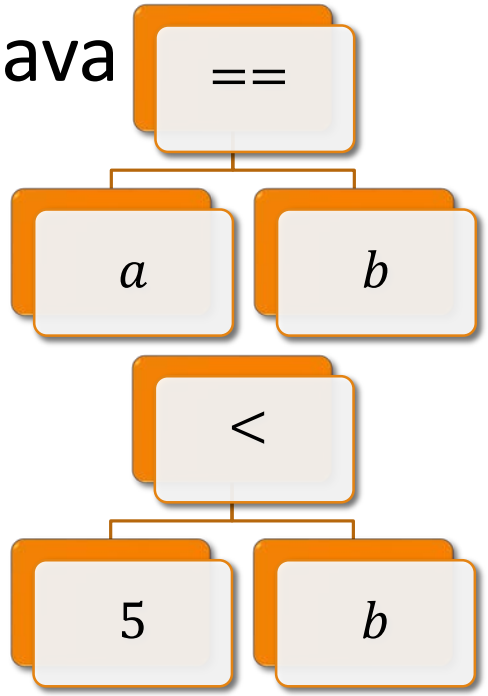
Bin.java



- Só deve ser aplicado a tipos numéricos (inteiro ou real)
- O tipo do operador vai depender dos seus operandos
- DL permite operar tipos reais com inteiros
- **Teste o operador binário!**

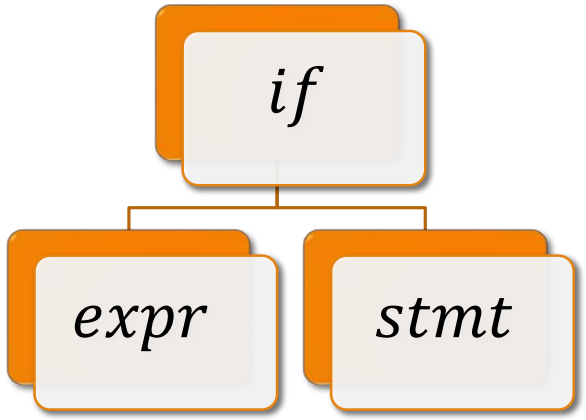
```
public Bin( Token op, Expr e1, Expr e2 ) {
    super(op, null);
    type = maxType( e1.type(), e2.type() );
    if ( this.type == null )
        error("tipos incompatíveis");
    expr1 = e1;
    expr2 = e2;
    addChild(expr1);
    addChild(expr2);
}
```

Rel.java



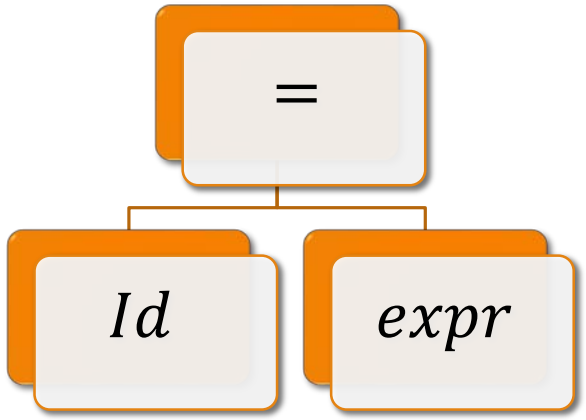
- A igualdade pode ser aplicada entre operadores lógicos e entre numéricos
- As comparações $<$ e $<=$ só podem ser aplicadas entre tipos numéricos
- O resultado é sempre booleano
- **Como fica o código?**

If.java



- No comando If é esperado que $expr$ seja do tipo booleano
- **Como fica o código?**

Assign.java



- Em uma atribuição é esperado que o tipo da variável seja igual ao valor que ele irá receber
- **Como fica o código?**

Write.java

- Vamos considerar que o comando `Write` só possa imprimir o valor de variáveis numéricas
- **Como fica o código?**

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.