SBVCONC: Construção de Compiladores

Aula 08: Árvores Sintáticas Abstratas

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Árvores Sintáticas Abstratas

- Modificaremos nosso *parser* mais uma vez, fazendo com que durante a análise do código fonte seja gerada uma representação intermediária do programa, chamada de árvore sintática abstrata (AST, *Abstract Syntax Tree*);
- Uma árvore sintática abstrata é similar à árvore de análise, mas sem os símbolos terminais e não-terminais;
- As árvores sintáticas abstratas fornecem uma representação explícita da estrutura do código fonte que pode ser usada para:
  - Análise de restrições adicional, por exemplo, restrições de tipos;
  - Algumas otimizações através de transformações na árvore;
  - Geração de código.

Representando as Árvores Sintáticas Abstratas

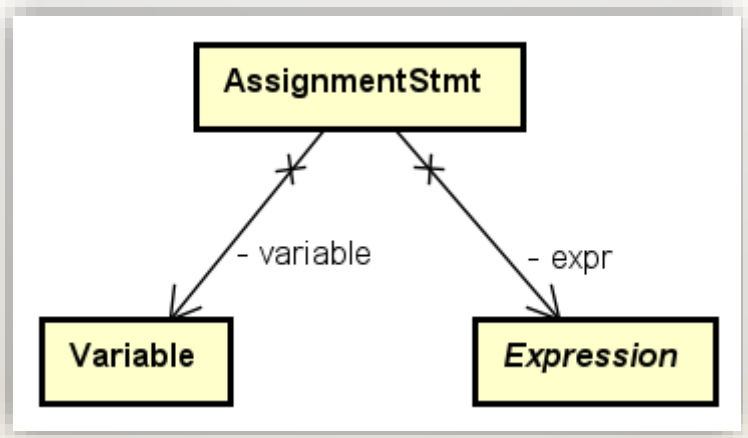
- Usaremos classes diferentes para representar os diferentes nós da nossa árvore sintática abstrata. Alguns exemplos:
  - `Program, ProcedureDecl, AssignmentStmt, LoopStmt, Variable, Expression...`
- Cada classe da AST possui variáveis de instância (campos) que referenciam seus filhos. Essas variáveis de instância que fornecerão a estrutura da árvore;
- Ocasionalmente, também incluiremos campos adicionais que suportarão o tratamento de erros, como a posição do *token* e a geração de código;
- Nossa AST pode também ser chamada de árvore sintática abstrata irregular e heterogênea.

Árvores Sintáticas Abstratas
Exemplo 1

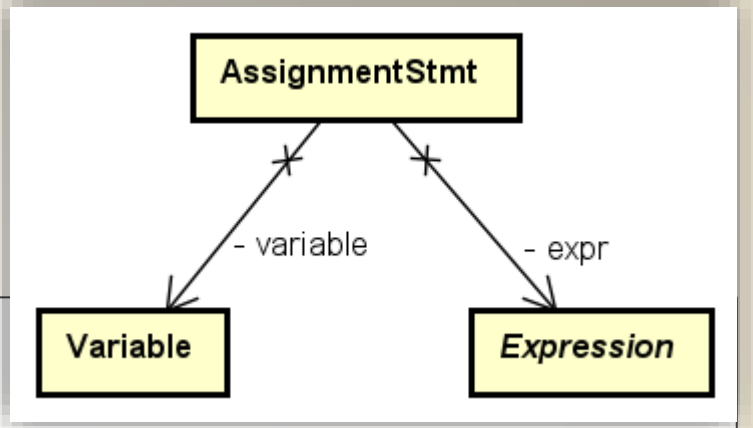
- Considere a regra abaixo para a instrução de atribuição:

```
assignmentStmt = variable ":=" expression ";" .
```

- As partes importantes das instruções de atribuição são:
  - variável (lado esquerdo da atribuição);
  - expressão (lado direito da atribuição);
- Criaremos um nó da AST para uma instrução de atribuição usando a seguinte estrutura:



Classe AssignmentStmt



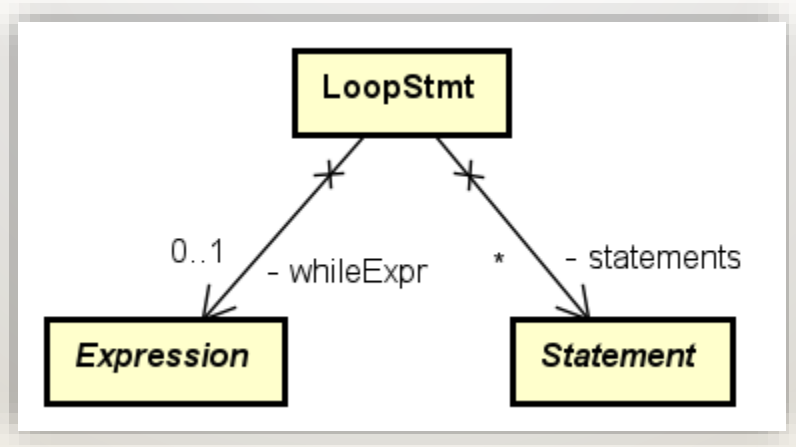

```
public class AssignmentStmt extends Statement {

    private Variable variable;
    private Expression expr;

    // posição do operador de atribuição, usada para reportar erro
    private Position assignPosition;

    public AssignmentStmt( Variable variable,
                           Expression expr,
                           Position assignPosition ) {
        this.variable = variable;
        this.expr = expr;
        this.assignPosition = assignPosition;
    }

    ...

}
```

Árvores Sintáticas Abstratas
Exemplo 2

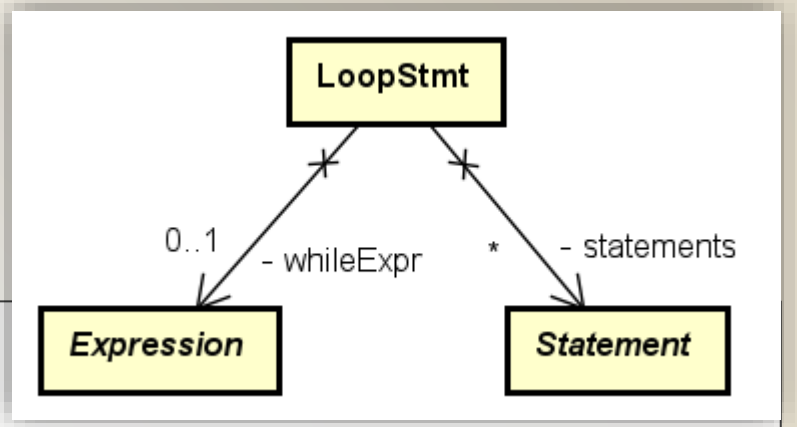
- Considere a regra abaixo para a instrução de laço:

```
loopStmt = ( "while" booleanExpr )?
                "loop" statements "end" "loop" ";" .
```

- Assim que uma instrução de laço tiver sido analisada, não precisamos manter os símbolos não-terminais. A AST para uma instrução de laço deve conter apenas as instruções do corpo do laço e a expressão booleana opcional, que caso não exista, será nula.



Classe LoopStmt

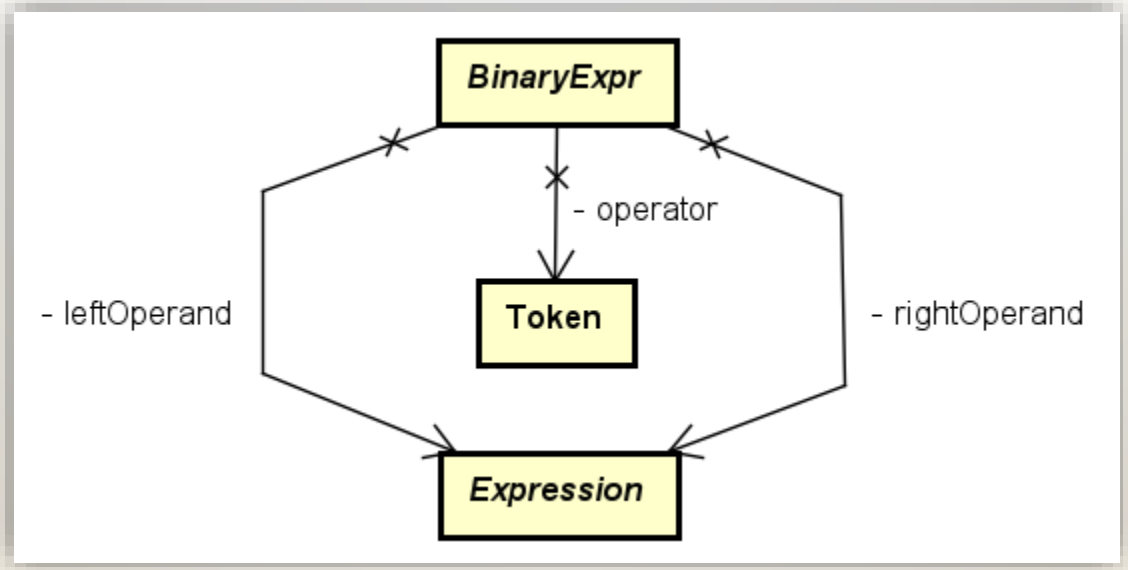



```
public class LoopStmt extends Statement {

    private Expression whileExpr;
    private List<Statement> statements;

    public LoopStmt( Expression whileExpr,
                     List<Statement> statements ) {
        this.whileExpr  = whileExpr;
        this.statements = statements;
        ...
    }

    ...

}
```

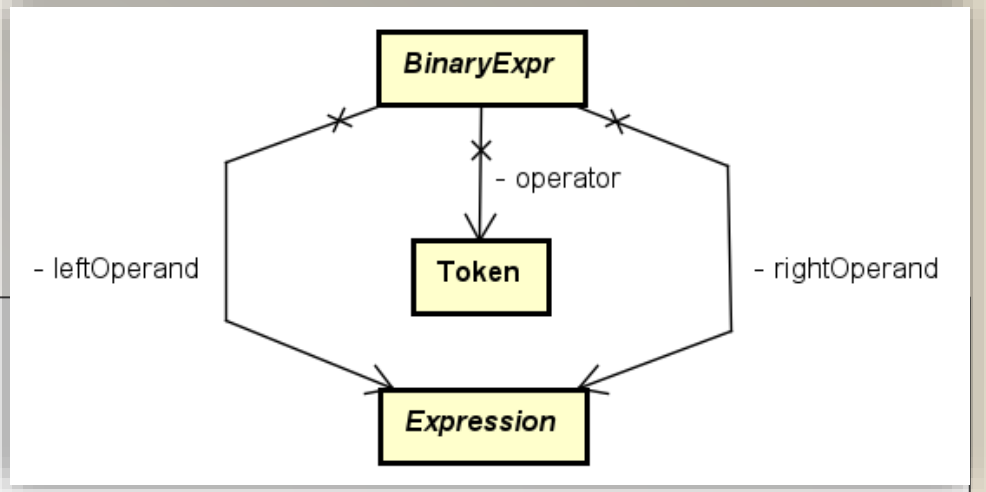
Pode ser null!

Árvores Sintáticas Abstratas
Exemplo 3

- Para expressões binárias, parte da gramática existe simplesmente para definir a precedência dos operadores;
- Assim que uma expressão tiver sido analisada, não precisamos preservar informações adicionais sobre não-terminais que foram introduzidas para definir precedência (`relation`, `simpleExpr`, `term`, `factor` etc.);
- A AST para uma expressão binária deve conter somente o operador e os operandos do lado esquerdo e direito. O algoritmo de análise deve construir a AST de modo a preservar a precedência dos operadores.



Classe BinaryExpr




```
public abstract class BinaryExpr extends Expression {

    private Expression leftOperand;
    private Token operator;
    private Expression rightOperand;

    public BinaryExpr( Expression leftOperand,
                       Token operator,
                       Expression rightOperand ) {
        this.leftOperand  = leftOperand;
        this.operator     = operator;
        this.rightOperand = rightOperand;
        ...
    }

    ...

}
```

A Estrutura de Uma Árvore Sintática Abstrata

- Há uma classe abstrata chamada **AST** que serve como superclasse para todas as classes que serão usadas para construir as árvores sintáticas abstratas;
- A classe **AST** contém a implementação dos métodos comuns à todas as subclasses, além das declarações de métodos abstratos que serão requeridos nas subclasses concretas;
- Todas as classes da AST serão definidas no subpacote `"...ast"`.

Esboço da Classe AST

```
public abstract class AST {

    ...

    // Verifica as restrições semânticas/contextuais.
    public abstract void checkConstraints();

    // Emite o código objeto para a AST
    public abstract void emit() throws CodeGenException, IOException;

}
```

Os métodos `checkConstraints()` e `emit()` provêm um mecanismo para caminharmos na estrutura da árvore usando chamadas recursivas aos nós subordinados.

Subclasses de `AST`

- Criaremos uma hierarquia de classes, sendo algumas delas algumas abstratas, que serão subclasses diretas ou indiretas de `AST`;
- Cada nó da AST construída pelo *parser* será um objeto de alguma classe da hierarquia de `AST`;
- A maioria das classes da hierarquia corresponderá e terá nomes similares aos símbolos não-terminais da gramática, mas nem todas as ASTs terão essa propriedade. Por exemplo, nos slides anteriores já discutimos sobre as expressões binárias. Não precisamos de classes para modelar os não-terminais `simpleExpr`, `term`, `factor` etc.

Usando as Classes de Coleção

- Alguns métodos de análise retornarão simplesmente uma lista de objetos da AST;
- **Exemplos:**

```
public List<InitialDecl> parseInitialDecls() throws IOException
public List<SubprogramDecl> parseSubprogramDecls() throws IOException
public List<Token> parseIdentifiers() throws IOException
public List<Statement> parseStatements() throws IOException
public List<ParameterDecl> parseFormalParameters() throws IOException
public List<Expression> parseActualParameters() throws IOException
```

Convenções de Nomenclatura para ASTs

- A maioria das classes da AST têm nomes similares aos não-terminais da gramática:
  - `Program, FunctionDecl, AssignmentStmt, LoopStmt...`
- O método de análise para um não-terminal gerará um objeto correspondente da AST:
  - `parseProgram` retorna um objeto do tipo `Program`
  - `parseLoopStmt` retorna um objeto do tipo `LoopStmt`
  - etc.
- Métodos de análise com nomes no plural retornarão uma lista de objetos da AST:
  - A gramática foi escrita para possuir essa propriedade.

Convenções de Nomenclatura para ASTs

- **Exemplos:**

```
public abstract class Statement extends AST ...
public class LoopStmt extends Statement ...
```

- O método de análise `parseLoopStmt()` é responsável em criar um nó do tipo `LoopStmt` da AST. Ao invés de retornar `void`, esse método passará a retornar um objeto da classe `LoopStmt`;
- De forma parecida, o método de análise `parseStatements()` retornará uma lista de objetos do tipo `Statement`, onde cada objeto `Statement` será um `AssignmentStmt`, ou um `LoopStmt`, ou um `IfStmt` etc.

O Método `parseLiteral()`

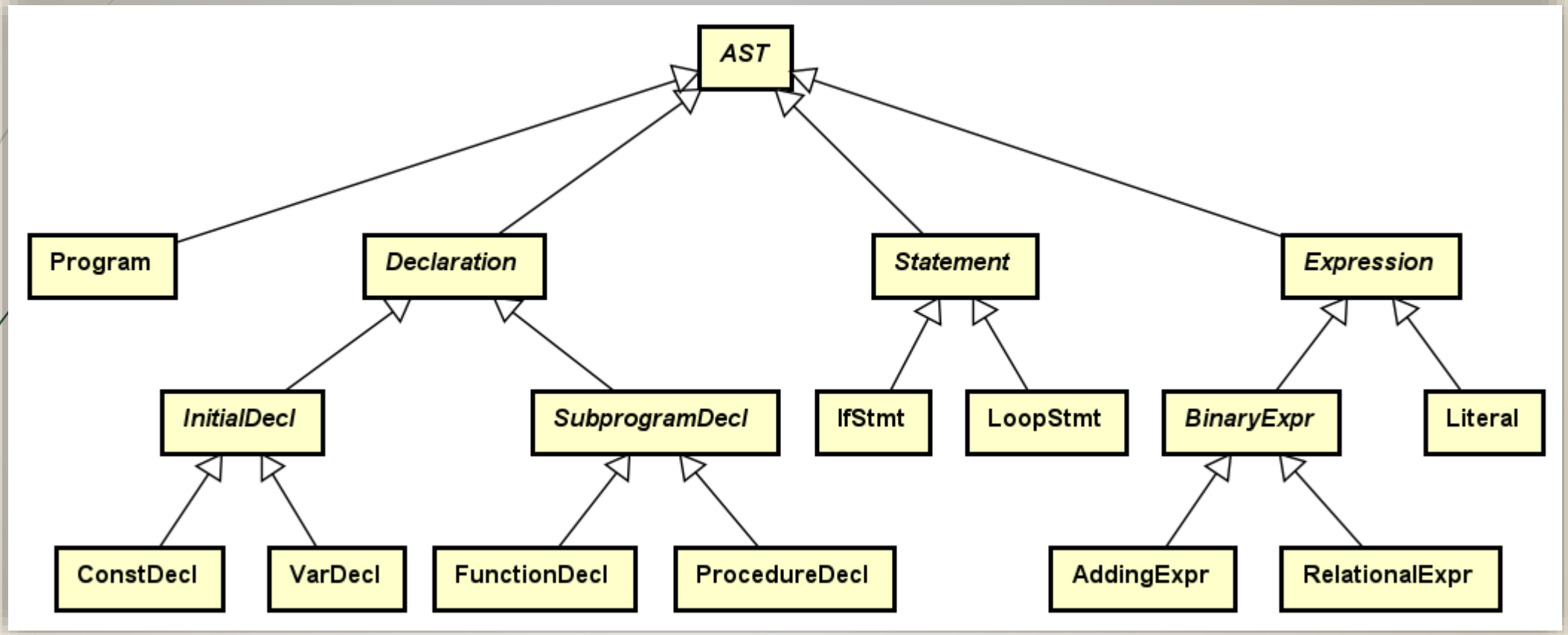
- O método `parseLiteral()` é um caso especial!
- Dado que os literais são *tokens* retornados pelo scanner, o método `parseLiteral()` retornará simplesmente objetos do tipo `Token`. Não há uma classe da AST com o nome de `Literal`;
- Regras relevantes da gramática:

```
        literal = intLiteral | charLiteral | stringLiteral
                | booleanLiteral .
booleanLiteral = "true" | "false" .
```

- **Método:**

```
public Token parseLiteral() throws IOException {
    ...
}
```

Diagrama Parcial das Classes das ASTs para a Linguagem CPRL



Restrições da Linguagem Associadas aos Identificadores

- Um *parser* construído somente com esse conjunto de regras de análise, não rejeitará programas que violam certas restrições da linguagem como "um identificador deve ser declarado apenas uma vez";
- **Exemplos:** sintaxe válida, mas inválida em relação às restrições contextuais.

```
var x : Integer;
begin
   y := 5;
end.
```

```
var c : Char;
begin
   c := -3;
end.
```

Classe `IdTable`

- Estenderemos a classe `IdTable` para nos ajudar a acompanhar não somente os tipos dos identificadores que foram declarados, mas também suas declarações;
- A classe `Declaration` é parte da hierarquia das ASTs. Um objeto desse tipo contém uma referência ao *token* do identificador e informações sobre seu tipo. Usaremos diferentes subclasses de `Declaration` para os vários tipos de declarações, por exemplo `ConstDecl`, `VarDecl`, `ProcedureDecl` etc.

Métodos Importantes da Classe `IdTable`

```
/**
 * Retorna o nível de escopo atual.
 */
public ScopeLevel getScopeLevel()

/**
 * Abre um novo escopo para identificadores.
 */
public void openScope()

/**
 * Fecha o escopo mais interno.
 */
public void closeScope()
```

`ScopeLevel` é uma enumeração com apenas dois valores: `PROGRAM` e `SUBPROGRAM`.

Métodos Importantes da Classe `IdTable`

```
/**
 * Insere uma declaração no nível de escopo atual.
 * @throws ParserException se o token associado à declaração
 * já tiver sido definido dentro do escopo atual.
 */
public void add( Declaration decl ) throws ParserException

/**
 * Retorna a Declaration associada ao texto do token do identificador.
 * Returna null se o identificador não for encontrado.
 * Esse método busca em escopos mais externos caso necessário.
 */
public Declaration get( Token idToken )
```

Inserindo Declarações na `IdTable`

- Quando um identificador é declarado, o *parser* tentará adicionar sua declaração à tabela dentro do escopo atual. Note que a declaração já contém o *token* do identificador;
  - Lança uma exceção caso uma declaração de mesmo nome (mesmo texto do *token*) já tiver sido declarada previamente no escopo atual.
- **Exemplo** (dentro do método `parseConstDecl()`):

```
Token constId = scanner.getToken();
...
constDecl = new ConstDecl( constId, constType, literal );
idTable.add( constDecl );
```

Lança uma `ParserException` se o *token* do identificador `constId` já estiver definido no escopo atual.

Interface `NamedDecl`

- Os identificadores declarados usando `VarDecl` (que será convertida numa lista de `SingleVarDecl`, processo que será descrito posteriormente), ou `ParameterDecl` têm usos similares dentro da CPRL:

```
x := y;
```

- A variável `x` pode não ter sido declarada numa declaração de variável ou de parâmetro:
  - O mesmo se aplica ao valor nomeado `y`
- É necessário tratar ambos os tipos de declaração de forma uniforme em vários pontos durante a análise, sendo que alcançaremos isso criando a interface `NamedDecl` e especificando que `SingleVarDecl` e `ParameterDecl` implementam essa interface.

Interface `NamedDecl`

- Cinco métodos importantes da interface `NamedDecl`:

```
public Type getType();
public int getSize();
public ScopeLevel getScopeLevel();
public void setRelAddr( int relAddr );
public int getRelAddr();
```

Usando a Interface `NamedDecl`

```
// código retirado de parseStatement()

if ( symbol == Symbol.identifier ) {

    Declaration decl = idTable.get( scanner.getToken() );

    if ( decl != null ) {

        if ( decl instanceof NamedDecl ) {
            stmt = parseAssignmentStmt();
        }
        ...

    }

}

...
```

Usando a `IdTable` para Verificar Ocorrências Aplicadas dos Identificadores

- Quando um identificador é encontrado na parte de instruções de um programa ou subprograma, por exemplo, com parte de uma expressão ou chamada de subprograma, o *parser* vai:
  - Verificar se o identificador foi encontrado;
  - Usar a informação de como o identificador foi declarado para facilitar a análise correta, por exemplo, você não pode atribuir um valor a um identificador que foi declarado como constante.

Usando a `IdTable` para Verificar Ocorrências Aplicadas dos Identificadores

- Exemplo (dentro do método `parseVariableExpr()`):

```
Token idToken = scanner.getToken();
match( Symbol.identifier );
Declaration decl = idTable.get( idToken );

if ( decl == null ) {
    throw error( "Identifier \"" + idToken
                 + "\" has not been declared.“ );
} else if ( !( decl instanceof NamedDecl ) ) {
    throw error( "Identifier \"" + idToken
                 + "\" is not a variable.“ );
}
```

Tipos na CPRL

- O compilador usa duas classes para prover suporte aos tipos da CPRL;
- A classe `Type` encapsula os tipos da linguagem e seus tamanhos:
  - Tipos predefinidos são declarados como constantes estáticas;
  - A classe `Type` também contém um método estático que retorna o tipo de um símbolo de literal:

```
public static Type getTypeOf( Symbol literal )
```

- A classe `ArrayType` estende `Type` para prover suporte adicional aos arrays.

Classe Type

- A classe `Type` encapsula os tipos da linguagem e seus tamanhos (número de bytes) para a CPRL;
- Os tamanhos dos tipos são inicializados com valores apropriados à máquina virtual da CPRL:
  - 4 para `Integer`, 2 para `Character`, 1 para `Boolean` etc.
- Os tipos predefinidos são declarados como constantes estáticas:

```
public static final Type Boolean = new Type(...);
public static final Type Integer = new Type(...);
public static final Type Char    = new Type(...);
public static final Type String  = new Type(...);
public static final Type Address = new Type(...);
public static final Type UNKNOWN = new Type(...);
```

Classe ArrayType

- A classe `ArrayType` estende a classe `Type`;
  - Sendo assim, os tipos de arrays também são tipos;
- Além do tamanho total do array (em bytes), a classe `ArrayType` também acompanha a quantidade e o tipo dos elementos:

```
/**
 * Constrói um tipo array com o nome especificado, a quantidade
 * de elementos e o tipo dos elementos contidos no mesmo.
 */
public ArrayType( String typeName,
                  int numElements,
                  Type elementType )
```

- Quando o *parser* analisa uma declaração de tipo de array, o construtor da classe `ArrayTypeDecl` das ASTs cria um objeto do tipo `ArrayType`.

Analisando uma ConstDecl

```
/**
 * Analisa a regra gramatical abaixo:
 *
 * constDecl = "const" constId ":=" literal ";" . *
 *
 * @return a declaração da constante analisada. Retorna
 * uma declaração nula se a análise falhar.
 */
public ConstDecl parseConstDecl() throws IOException {

  try {

    match( Symbol.constRW );
    Token constId = scanner.getToken();
    match( Symbol.identifier );
    match( Symbol.assign );
    Token literal = parseLiteral();
    match( Symbol.semicolon );

    Type constType = Type.UNKNOWN;
    if ( literal != null ) {
      constType = Type.getTypeOf( literal.getSymbol() );
    }
    ConstDecl constDecl = new ConstDecl( constId,
                                         constType,
                                         literal );
    idTable.add( constDecl );

    return constDecl;

  } catch ( ParserException e ) {

    ErrorHandler.getInstance().reportError( e );
    recover( initialDeclFollowers );

    return null;

  }

}
```

O Nível do Escopo de uma Declaração de Variável

- Durante a geração de código, quando uma variável ou um valor nomeado são referenciados na parte de instruções de um programa ou subprograma, precisamos ser capazes de determinar onde a variável foi declarada;
- A classe `IdTable` contém o método `getScopeLevel()` que retornará o nível de aninhamento de bloco do escopo atual:
  - `PROGRAM` para objetos declarados no escopo mais externo (programa);
  - `SUBPROGRAM` para objetos declarados dentro de subprogramas;
- Quando uma variável é **declarada**, a declaração é inicializada no nível corrente:

```
varDecl = new VarDecl( identifiers,
                       varType,
                       idTable.getScopeLevel() );
```

Níveis de Escopo
Exemplo

```
var x : Integer;   // o nível de escopo da declaração é PROGRAM
var y : Integer;   // o nível de escopo da declaração é PROGRAM

procedure p is       // o nível de escopo da declaração é PROGRAM
   var x : Integer;  // o nível de escopo da declaração é SUBPROGRAM
   var b : Integer;  // o nível de escopo da declaração é SUBPROGRAM
begin
   ... x ...   // x foi declarada no escopo SUBPROGRAM
   ... b ...   // b foi declarada no escopo SUBPROGRAM
   ... y ...   // y foi declarada no escopo PROGRAM
end p;

begin
    ... x ...     // x foi declarada no escopo PROGRAM
    ... y ...     // y foi declarada no escopo PROGRAM
    ... p ...     // p foi declarada no escopo PROGRAM
end.
```

VarDecl versus SingleVarDecl

- Uma declaração de variável pode declarar vários identificadores com o mesmo tipo:

```
var x, y, z : Integer;
```

- Essa declaração é logicamente equivalente a declarar cada variável separadamente:

```
var x : Integer;
var y : Integer;
var z : Integer;
```

- Para simplificar a verificação de restrições e geração de código, dentro da AST veremos uma declaração de variável como uma coleção de várias declarações únicas de variáveis.

Classe `SingleVarDecl`

```
public class SingleVarDecl extends InitialDecl implements NamedDecl {

    private ScopeLevel scopeLevel;
    ...

    public SingleVarDecl( Token identifier,
                          Type varType,
                          ScopeLevel scopeLevel ) {

        super( identifier, varType );
        this.scopeLevel = scopeLevel;

    }

    ...

}
```

Classe VarDecl

```
public class VarDecl extends InitialDecl {

    // a lista de SingleVarDecls para a declaração de variáveis
    private List<SingleVarDecl> singleVarDecls;

    public VarDecl( List<Token> identifiers,
                    Type varType,
                    ScopeLevel scopeLevel ) {

        super( null, varType );
        singleVarDecls = new ArrayList<>( identifiers.size() );

        for ( Token id : identifiers ) {
            singleVarDecls.add( new SingleVarDecl( id, varType, scopeLevel) );
        }

    }
    ...
}
```

Uma `VarDecl` é simplesmente uma lista de `SingleVarDecls`.

O Método `parseInitialDecls()`

- O método `parseInitialDecls()` constrói/retorna uma lista de declarações iniciais;
- Para as declarações de constantes e tipos de arrays, esse método simplesmente os adiciona na lista;
- Para a declaração de variáveis (`VarDecls`), esse método extrai as declarações de variáveis únicas (`SingleVarDecls`) e as adiciona na lista. A `VarDecl` original não é mais usada nesse ponto.

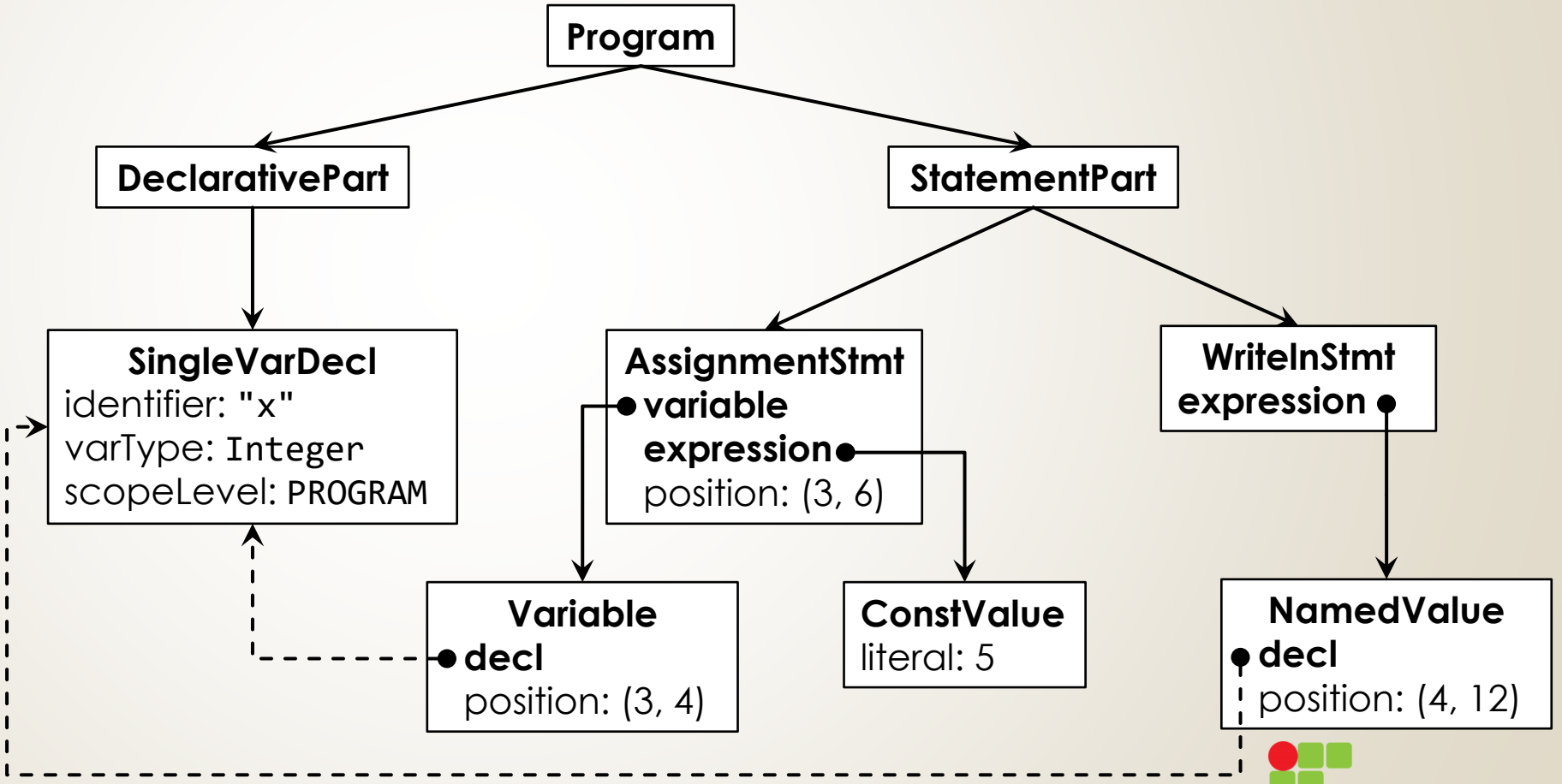
```
...
InitialDecl decl = parseInitialDecl();

if ( decl instanceof VarDecl ) {
    // adiciona as declarações de variáveis únicas
    VarDecl varDecl = (VarDecl) decl;
    for ( SingleVarDecl singleVarDecl : varDecl.getSingleVarDecls() ) {
        initialDecls.add( singleVarDecl );
    }
} else {
    initialDecls.add( decl );
}
```

Referências Estruturais versus Referências Não-Estruturais

- A maioria dos campos das classes da AST representam referências estruturais que correspondem às arestas da árvore:
  - A classe `Program` tem uma referência à sua parte declarativa e à sua parte de instruções;
  - A classe `BinaryExpr` tem referências ao operando da esquerda, ao operador e ao operando da direita;
- Alguma classes da AST têm campos que não correspondem às arestas da árvore:
  - A classe `Variable` tem uma referência de volta à sua declaração, da mesma forma que na classe `NamedValue`;
  - A classe `ExitStmt` tem uma referência à instrução de laço que a contém;
- Essas referências não-estruturais são usadas durante a análise de restrições e a geração de código.

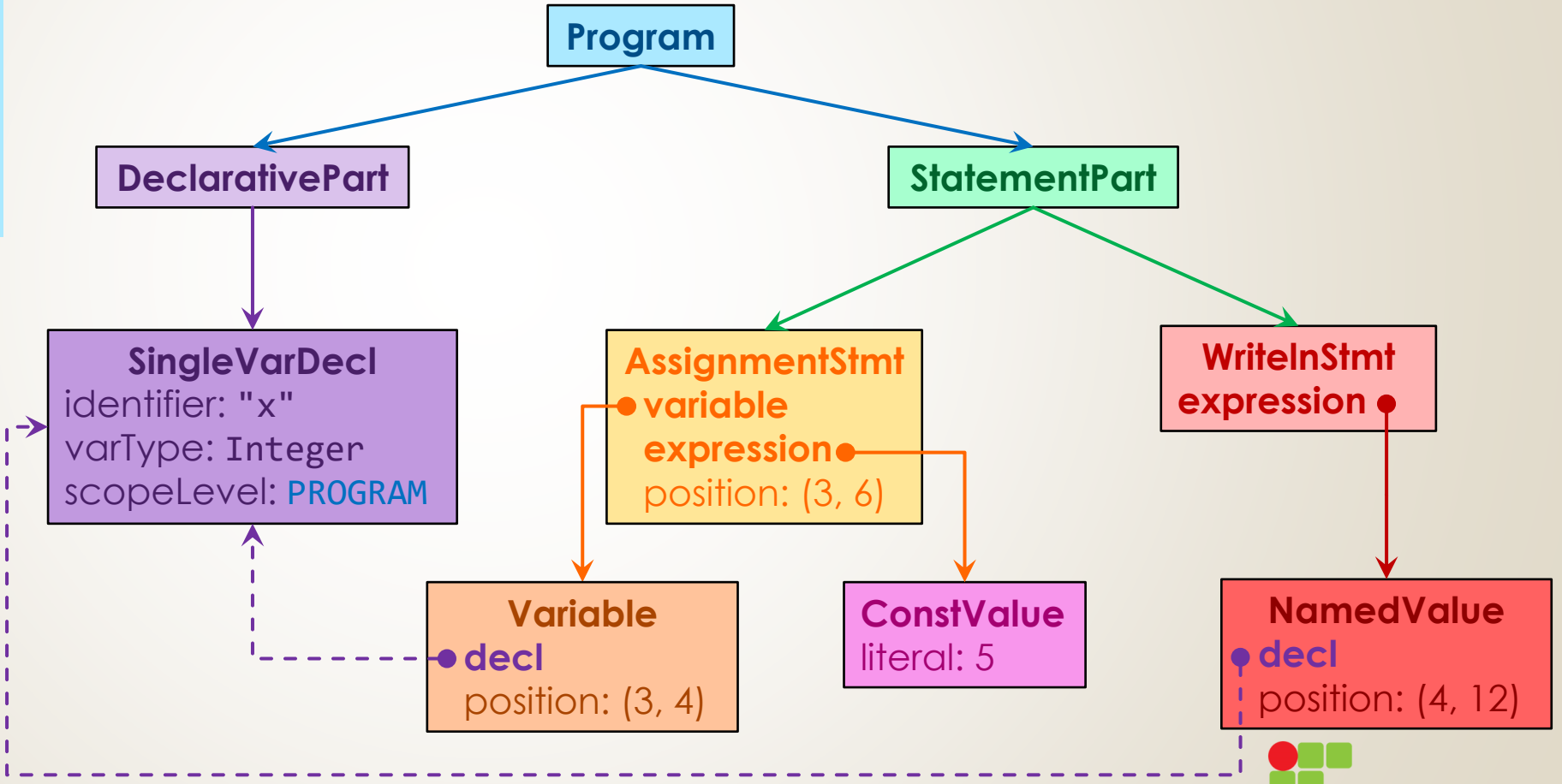
Exemplo de Uma AST

```
var x : Integer;
begin
   x := 5;
   writeln x;
end.
```



Exemplo de Uma AST

```
var x : Integer;
begin
    x := 5;
    writeln x;
end.
```



Determinando o Tipo das Expressões

- Dado que a CPRL tem tipagem estática é possível determinar o tipo de todas expressões em tempo de compilação e a classe `Expression` da AST tem uma propriedade usada para indicar o tipo da expressão, sendo que esta propriedade é herdada por todas as suas subclasses;
- Em geral, determinaremos o tipo de uma expressão no construtor da classe da expressão da AST.

Exemplo: `RelationalExpr`

- Uma expressão relacional é uma expressão binária em que o operador é um operador relacional como "<=" ou ">";
- Independentemente dos tipos dos dois operandos, uma expressão relacional sempre tem o tipo `Boolean`;
- Construtor da classe `RelationalExpr`:

```
public RelationalExpr( Expression leftOperand,
                       Token operator,
                       Expression rightOperand ) {
    super( leftOperand, operator, rightOperand );
    setType( Type.Boolean );
    ...
}
```

Exemplo: `AddingExpr`

- Para a maioria das linguagens de programação "reais", determinar o tipo de uma expressão de adição pode ser um pouco complicado;
- As linguagens C e Java possuem vários tipos numéricos com diversas regras conversão automática (coerções e promoções) quando um operador tem diferentes tipos de operandos;
- Na CPRL, uma expressão de adição sempre é do tipo `Integer`, da mesma forma que uma expressão de multiplicação.
- Construtor da classe `AddingExpr`:

```
public AddingExpr( Expression leftOperand,
                   Token operator,
                   Expression rightOperand ) {
    super( leftOperand, operator, rightOperand );
    setType( Type.Integer );
    ...
}
```

Exemplo: `Variable`

- O tipo de uma variável, bem como para um valor nomeado, é inicializado com o tipo especificado em suas declarações;
- Construtor para a classe `Variable`:

```
public Variable( NamedDecl decl,
                 Position position,
                 List<Expression> indexExprs ) {
    super( decl.getType(), position );
    this.decl = decl;
    this.indexExprs = indexExprs;
}
```

Exemplo: `Variable`

- O tipo utilizado na inicialização das variáveis é correto para os tipos predefinidos como `Integer` e `Char`, mas para os arrays é necessário trabalho adicional;
- Considere as seguintes declarações:

```
type T1 is array(10) of Integer;
type T2 is array(10) of T1;
var a, b : T2;
```

- Enquanto o tipo declarado (inicializado) tanto para `a` quanto para `b` é `T2`, podemos ter uma variável ou um valor nomeado com zero, um ou duas expressões para os índices como a seguir:

```
a := b;              // o tipo da variável e do valor nomeado é T2
a[0] := b[0];        // o tipo da variável e do valor nomeado é T1
a[1][6] := b[5][7];  // o tipo da variável e do valor nomeado é Integer
```

Exemplo: `Variable`

- Para os arrays, determinamos o tipo correto da variável ou do valor nomeado no método `checkConstraints()`.

```
for ( Expression expr : indexExprs ) {


    expr.checkConstraints();

    if ( expr.getType() != Type.Integer ) {
        throw error(...);
    }

    if ( getType() instanceof ArrayType ) {
        ArrayType type = ( ArrayType ) getType();
        setType( type.getElementType() );
    } else {
        throw error(...);
    }

}
```

Mantendo o Contexto Durante a Análise Sintática

- Certas instruções da CPRL precisam ter acesso ao contexto em que elas estão incluídas, visando a verificação das restrições e a geração de código;
- Exemplo: `exit when n > 10;`
- As instruções `exit` tem significado somente quando estão aninhadas à laços e para a geração de código das mesmas é necessário saber qual laço que as incluem;
- De forma similar, a instrução `return` precisa saber de qual subprograma ela está retornando;
- As classes `LoopContext` e `SubprogramContext` serão usadas para manter a informação contextual nesses casos.

Classe LoopContext

```
/**
 * Retorna a instrução de laço que está sendo atualmente analisada.
 * Retorna null se não houver uma instrução de laço.
 */
public LoopStmt getLoopStmt()

/**
 * Chamado no início da análise de uma instrução de laço.
 */
public void beginLoop( LoopStmt stmt )

/**
 * Chamado ao se terminar a análise de uma instrução de laço.
 */
public void endLoop()
```

Classe SubprogramContext

```
/**
 * Retorna a declaração de subprograma que está sendo atualmente analisada.
 * Retorna null se não houver tal procedimento.
 */
public SubprogramDecl getSubprogramDecl()

/**
 * Chamado no início da análise de uma declaração de subprograma.
 */
public void beginSubprogramDecl( SubprogramDecl subprogDecl )

/**
 * Chamado ao se terminar a análise de uma declaração de subprograma.
 */
public void endSubprogramDecl()
```

Usando o Contexto Durante a Análise Síntática
Exemplo

- Quando se está analisando uma instrução de laço:

```
LoopStmt stmt = new LoopStmt();
...
loopContext.beginLoop( stmt );
stmt.setStatements( parseStatements() );
loopContext.endLoop();
```

- Quando se está analisando uma instrução `exit`:

```
LoopStmt loopStmt = loopContext.getLoopStmt();
if ( loopStmt == null ) {
    throw error( exitPosition,
                 "Exit statement is not nested within a loop." );
}
return new ExitStmt( expr, loopStmt );
```

Desenvolvimento do *Parser* da CPRL

Versão 3: Árvores Sintáticas Abstratas (**Projeto 4**)

- Inclua a geração da estrutura da AST, ou seja, os métodos de análise devem retornar objetos ou listas de objetos da AST, sendo que as classes serão fornecidas;
- Deixe vazio o corpo da implementação dos métodos abstratos `checkConstraints()` e `emit()` quando os sobrescrever;
- Use a nova versão da classe `IdTable` para checar os erros de escopo;
- Use a classe `LoopContext` para checar a instrução `exit` e a classe `SubprogramContext` para checar a instrução `return`;
- Nesse ponto da implementação, seu compilador deve aceitar todos os programas corretos/legais e rejeitar a maioria dos programas incorretos/ilegais. Alguns programas com erros relacionados à tipos ou erros "variados" ainda não serão rejeitados.

Bibliografia

MOORE JR., J. I. **Introduction to Compiler Design: an Object Oriented Approach Using Java**. 2. ed. [s.l.]:SoftMoore Consulting, 2020. 284 p.

AHO, A. V.; LAM, M. S.; SETHI, R. ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson, 2008. 634 p.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2. ed. Rio de Janeiro: Campus Elsevier, 2014. 656 p.

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo: Elsevier, 2016. 307 p.

SANTOS, P. R.; LANGOLOIS, T. **Compiladores: da teoria à prática**. Rio de Janeiro: LTC, 2018. 341 p.