SBVCONC: Construção de Compiladores

Aula 05: Análise Léxica (*Scanning*)

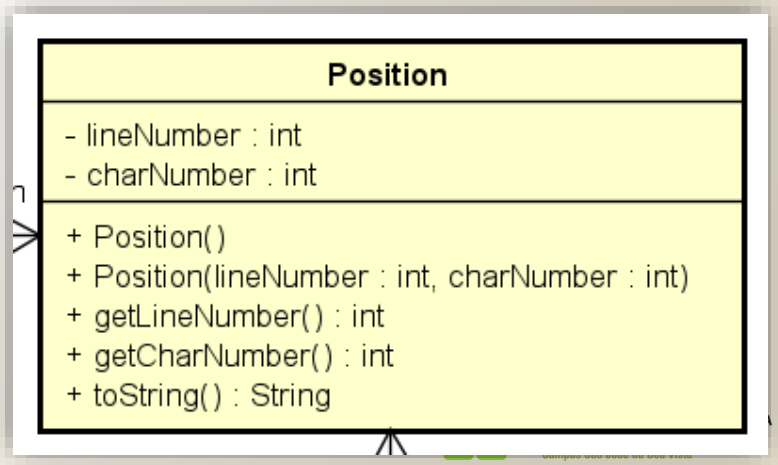
Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

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

Classe `Position`

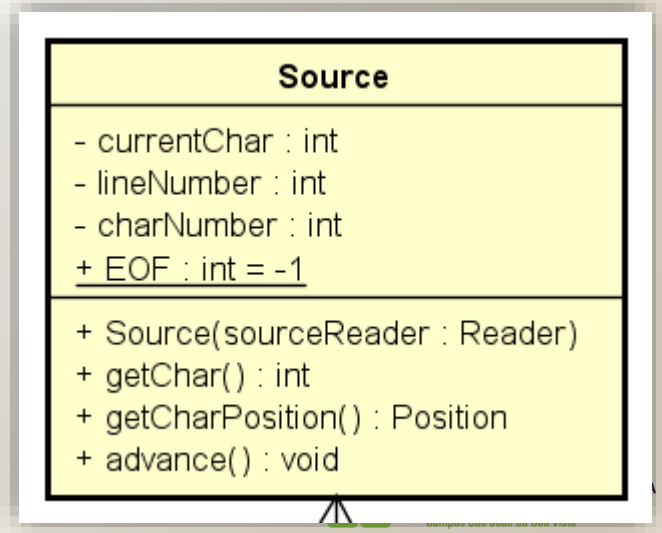
- A classe `Position` encapsula o conceito de uma posição dentro de um arquivo de código, sendo que essa informação é usada principalmente para relatar erros;
- A posição é caracterizada por um par ordenado de inteiros:
  - O número da linha relativa ao arquivo de código fonte;
  - O número do caractere (coluna) relativo à linha que o contém;
- Obs: Objetos do tipo `Position` são imutáveis, ou seja, depois de criados não podem ser modificados;
- Métodos chave:

```
public int getLineNumber()
public int getCharNumber()
```



Classe `Source`

- A classe `Source` é essencialmente um iterador de caminha através dos caracteres de um arquivo de código fonte, um caractere por vez. Em qualquer momento durante a iteração, pode-se examinar o caractere atual e sua posição dentro do arquivo fonte antes de avançar ao próximo caractere;
- Classe `Source`:
  - Encapsula o leitor de arquivos (*file reader*);
  - Mantém a posição de cada caractere no arquivo de código fonte;
  - Entrada: um `Reader` (normalmente um `FileReader`);
  - Saída: caracteres individuais e suas posições dentro do arquivo.



Classe Source: Métodos Chave

```
/**
 * Retorna o caractere atual (como um inteiro) dentro do arquivo fonte.
 * Retorna EOF se o fim do arquivo foi alcançado..
 */
public int getChar()

/**
 * Retorna a posição (número da linha e posição do caractere)
 * do caractere atual dentro do arquivo fonte.
 */
public Position getCharPosition()

/**
 * Avança ao próximo caractere dentro do arquivo fonte.
 */
public void advance() throws IOException
```

Como Testar a Classe Source

```
String fileName = args[0];
FileReader fileReader = new FileReader( fileName );
Source source = new Source( fileReader );

while ( source.getChar() != Source.EOF ) {

    int c = source.getChar();

    if ( c == '\n' ) {
        System.out.println();
    } else if ( c != '\r' ) {
        System.out.print( (char) c );
    }

    System.out.println( "\t" + source.getCharPosition() );
    source.advance();

}
```

Resultados do Teste da Classe `Source`

Arquivo de Entrada: `Source.java`

```
p   line 1, character 1
a   line 1, character 2
c   line 1, character 3
k   line 1, character 4
a   line 1, character 5
g   line 1, character 6
e   line 1, character 7
    line 1, character 8
e   line 1, character 9
d   line 1, character 10
u   line 1, character 11
.   line 1, character 12
c   line 1, character 13
i   line 1, character 14
t   line 1, character 15
a   line 1, character 16
...
```

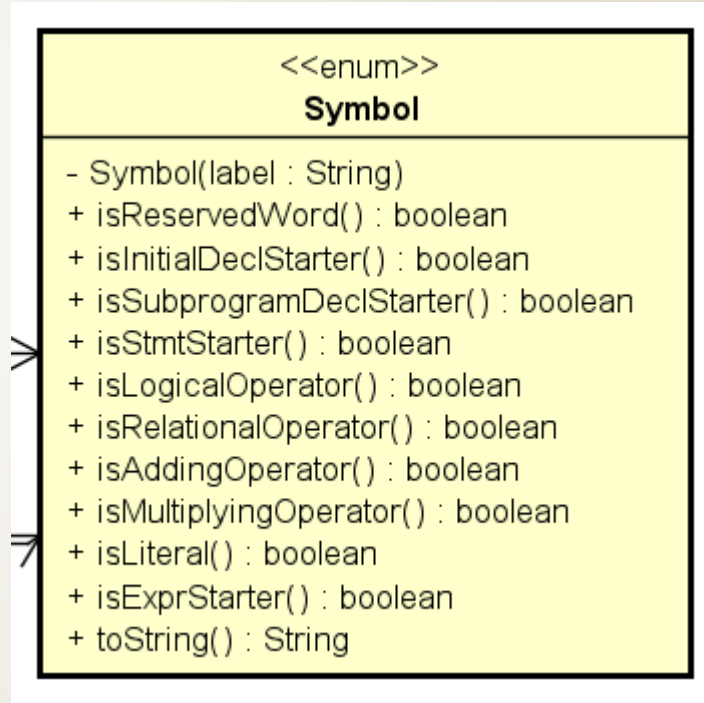
Symbol (Símbolo)
Tipo do Token (*Token Type*)

- O termo **symbol** (símbolo) será usado para nos referirmos às unidades léxicas básicas retornadas pelo `Scanner`. Da perspectiva do parser (analisador sintático) são esses os símbolos terminais;
- Os símbolos incluem:
  - Palavras reservadas (`while`, `if`, ...);
  - Operadores e pontuação (`:=`, `+`, `;`, ...);
  - Identificadores;
  - Literais de tipos;
  - Símbolos Especiais

Enumeração Symbol

```
public enum Symbol {

    // palavras reservadas
    BooleanRW("Boolean"),
    IntegerRW("Integer"),
    ...
    whileRW("while"),
    writeRW("write"),
    writelnRW("writeln"),

    // símbolos de operadores
    // aritméticos
    plus("+"),
    minus("-"),
    times("*"),
    divide("/"),

    ...
```

```
    ...

    // valores literais
    // e identificadores
    intLiteral("Integer Literal"),
    charLiteral("Character Literal"),
    stringLiteral("String Literal"),
    identifier("Identifier"),

    // símbolos especiais
    // de escaneamento
    EOF("End-of-File"),
    unknown("Unknown");

    ...
    // construtor e métodos
    // auxiliares

}
```



Token

- O termo **token** será usado para nos referirmos à combinação de símbolos com algumas informações adicionais, como:
  - A posição (número da linha e posição do caractere) do símbolo no arquivo fonte;
  - O texto associado ao símbolo;
- A informação adicional provida pelo token é usada para relatar erros, realizar análise de restrições e para geração de código, mas não para determinar se o programa é sintaticamente correto.

Exemplos: Texto Associado à Símbolos

- "`average`" para um identificador;
- "`100`" para um literal de inteiro;
- "`Hello, world.`" para um literal de String;
- "`while`" para a palavra chave `while`;
- "`<=`" para o operador menor ou igual;
- O texto associado à símbolos que são definidos pelo usuário (programador), como identificadores ou literais são mais significativos do que o texto associado à símbolos definidos pela linguagem, como palavras reservadas e operadores, que por sua vez não precisam ter seus textos armazenados.

Classe Token: Métodos Chave

```
/**
 * Retorna o símbolo do token.
 */
public Symbol getSymbol()

/**
 * Retorna a posição do token dentro do arquivo de código fonte.
 */
public Position getPosition()

/**
 * Retorna a representação em String do token.
 */
public String getText()
```

Implementando a Classe `Token`

A classe Token é implementada em duas classes separadas:

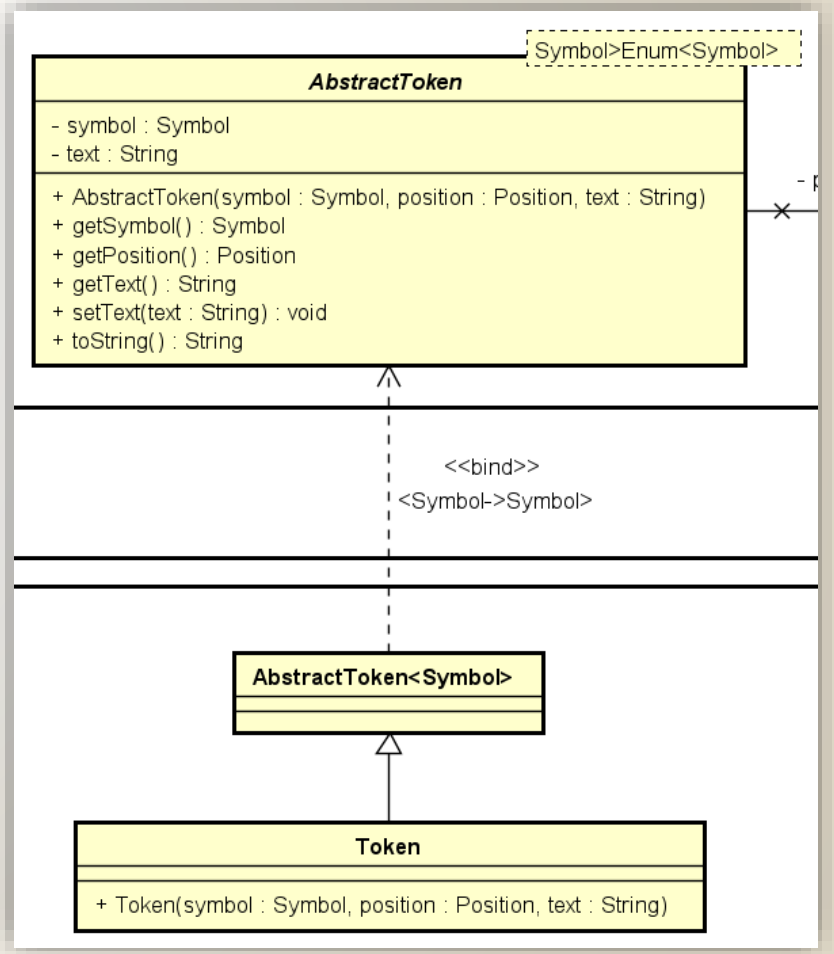
- Uma classe abstrata e genérica que pode ser instanciada com qualquer enumeração `Symbol`:

```
public abstract class AbstractToken
        <Symbol extends Enum<Symbol>>
```

- Uma classe concreta que fornece a enumeração `Symbol` da CPRL:

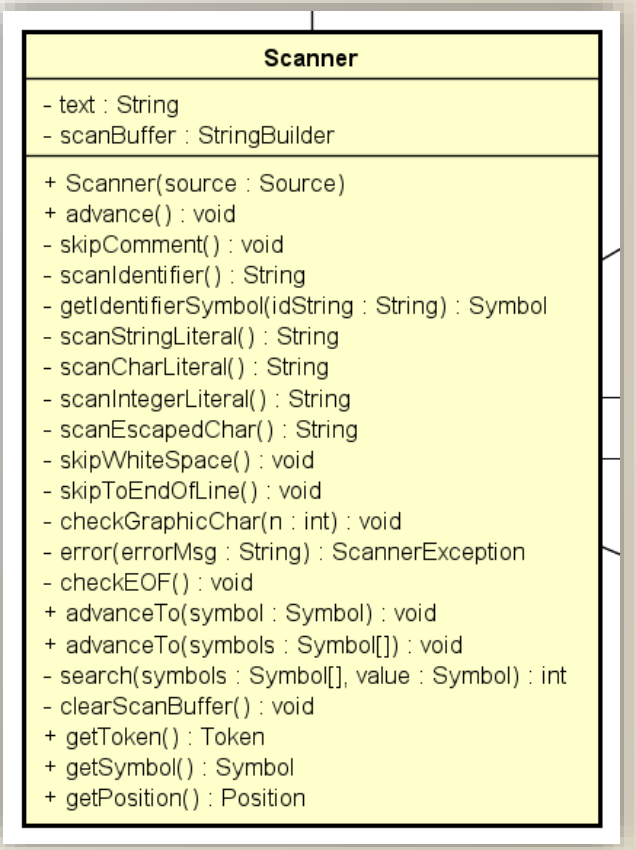
```
public class Token extends AbstractToken<Symbol>
```

- A classe `AbstractToken` é reusável em outros projetos de compiladores além do compilador da CPRL.



A Classe Scanner
Analisador Léxico

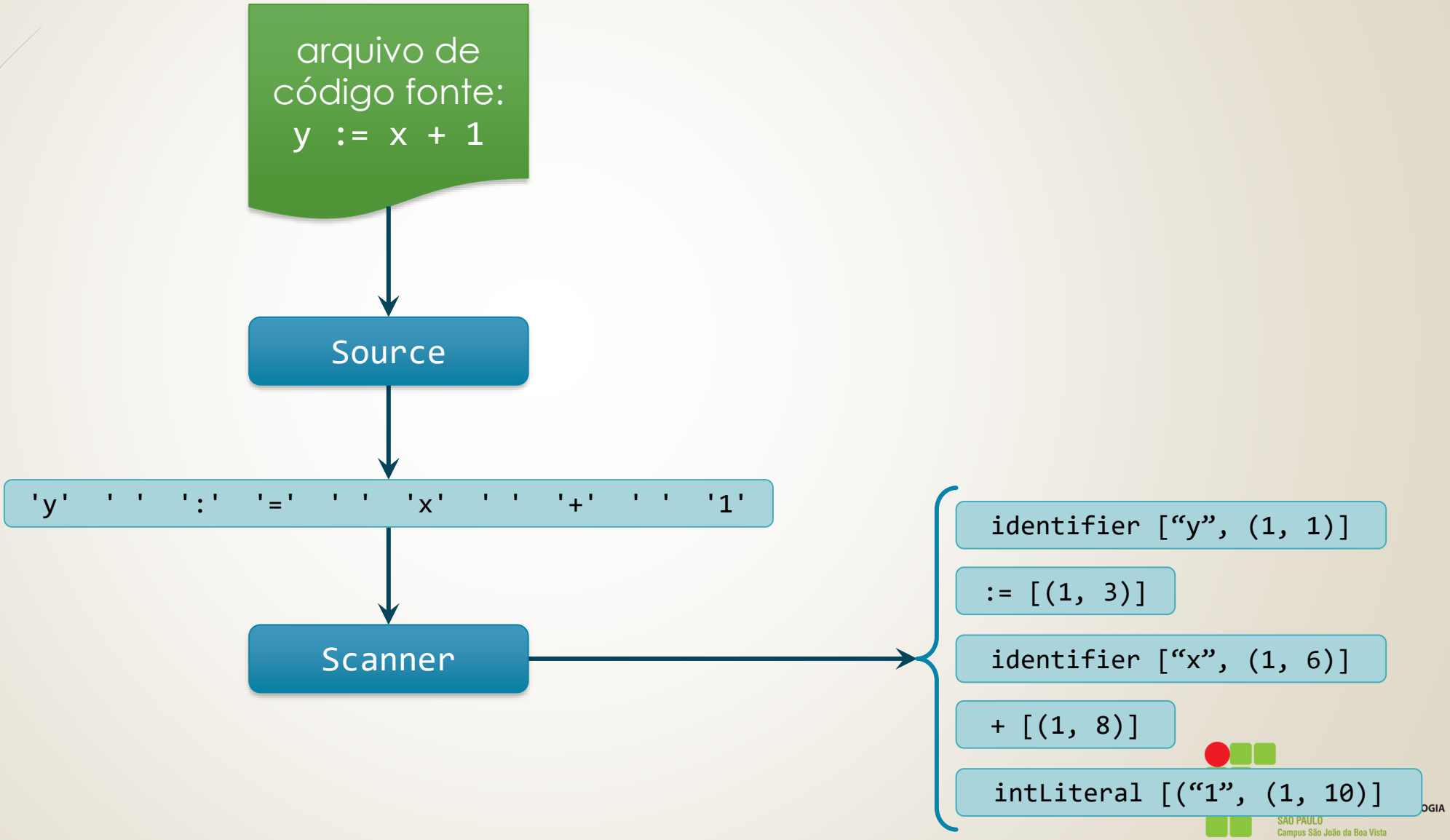
- A classe `Scanner` é essencialmente um iterador que caminha através dos tokens contidos no arquivo de código fonte, um token por vez. Em qualquer momento durante a iteração, podemos examinar o token atual, seu texto e sua posição dentro do arquivo fonte antes de avançar ao próximo token;
- Classe `Scanner`:
  - Consome os caracteres do arquivo de código fonte e constrói os tokens;
  - Remove espaços extra e comentários;
  - Reporta quaisquer erros;
  - Entrada: caracteres individuais vindos da classe `Source`;
  - Saída: os tokens que serão consumidos pelo parser.



Classe Scanner: Métodos Chave

```
/**
 * Retorna uma cópia do token atual contido no arquivo de código.
 */
public Token getToken()

/**
 * Retorna a referência do símbolo atual contido no arquivo de código.
 */
public Symbol getSymbol()

/**
 * Avança ao próximo token do arquivo de código.
 */
public void advance() throws IOException
```

Atuação das Classes Source e Scanner



Explicação dos Métodos da Classe Scanner

No Projeto no NetBeans

- `advance()`
- `scanIntegerLiteral()`
- `scanIdentifier()`
- `getIdentifierSymbol()`
- `scanStringLiteral()`

Erros Léxicos

- Há diversos tipos de erros que podem ser detectados pelo Scanner durante o processamento de um arquivo fonte. Exemplos incluem:
  - Falha ao fechar corretamente um literal de caractere ou um literal de string;
  - Encontrar um caractere que não inicia um símbolo válido como '#' or '@', etc;
- Método `error()`

```
private ScannerException error( String errorMsg ) {
    return new ScannerException( getPosition(), errorMsg );
}
```

Manipulando Erros Léxicos no Método advance()

```
...

} catch ( ScannerException e ) {

    // reporta o erro.
    // dentro do ErrorHandler foi mudada a forma de envio para usar
    // o stream de saída padrão da JVM, permitindo que nos testes
    // que vocês farão, as mensagens de erro também sejam verificadas
    // de modo a detectar a integridade do que foi implementado.
    ErrorHandler.getInstance().reportError( e );

    // configura o token como EOF ou unknown (desconhecido)
    symbol = source.getChar() == Source.EOF ? Symbol.EOF : Symbol.unknown;

}
```

Como Testar a Scanner

```
String fileName = args[0];
FileReader fileReader = new FileReader(fileName);

Source  source  = new Source(fileReader);
Scanner scanner = new Scanner(source);
Token   token;

do {
    token = scanner.getToken();
    printToken(token);
    scanner.advance();
} while ( token.getSymbol() != Symbol.EOF );
```

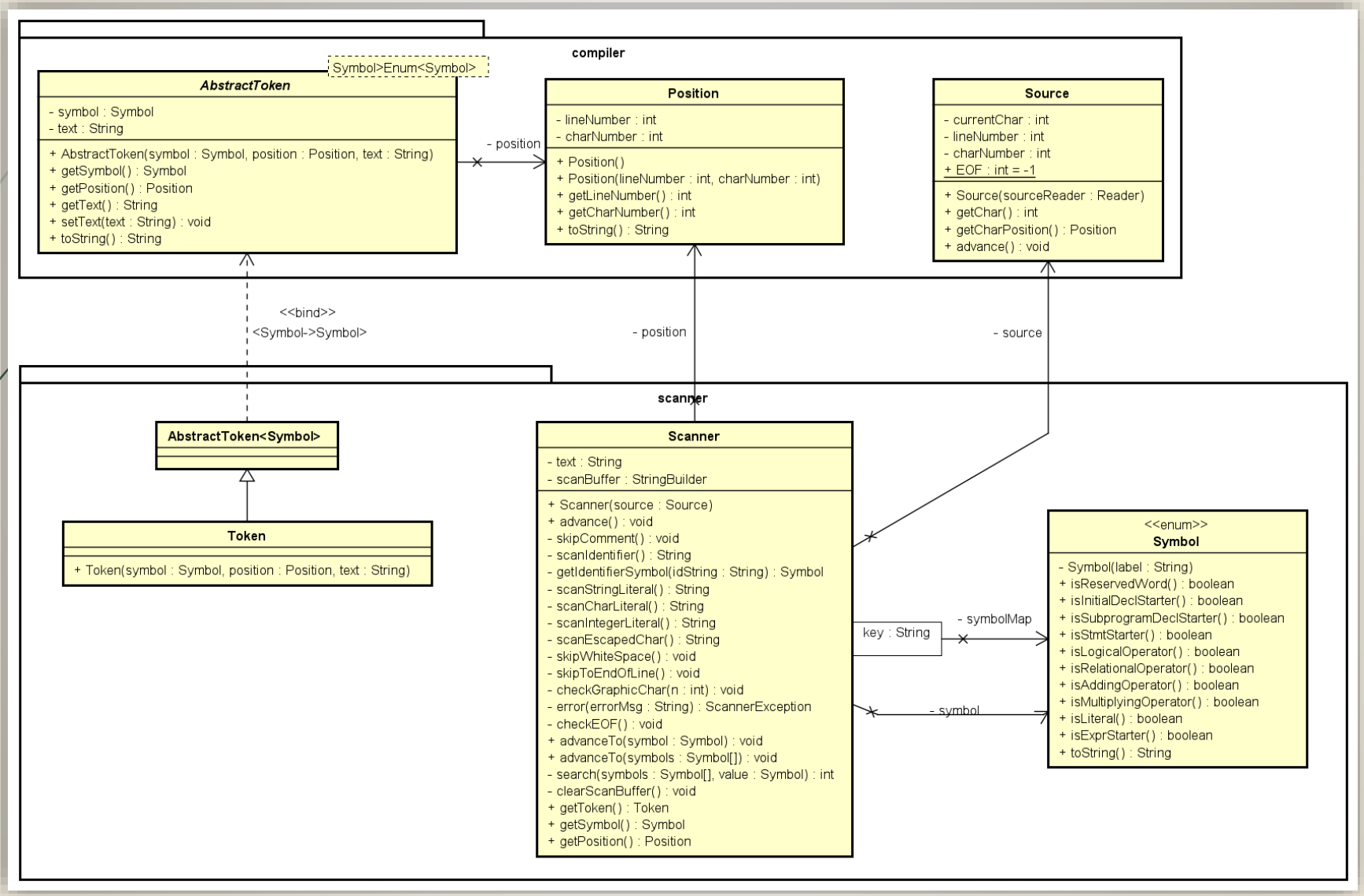
```
public static void printToken(Token token) {

    System.out.printf("line: %2d   char: %2d   token: ",
        token.getPosition().getLineNumber(),
        token.getPosition().getCharNumber());

    Symbol symbol = token.getSymbol();

    if (symbol.isReservedWord()) {
        System.out.print("Reserved Word -> ");
    } else if (   symbol == Symbol.identifier
              || symbol == Symbol.intLiteral
              || symbol == Symbol.stringLiteral
              || symbol == Symbol.charLiteral ) {
        System.out.print(token.getSymbol().toString() + " -> ");
    }

    System.out.println(token.getText());

}
```

Resultados do Teste da Classe `Scanner`

Arquivo de Entrada: `Correct_01.cprl` contido no diretório ScannerTests

```
line:  2   char:  1   token: Reserved Word -> and
line:  2   char: 11   token: Reserved Word -> array
line:  2   char: 21   token: Reserved Word -> begin
line:  2   char: 31   token: Reserved Word -> Boolean
...
line:  9   char: 31   token: Reserved Word -> while
line:  9   char: 41   token: Reserved Word -> write
line: 10   char:  1   token: Reserved Word -> writeln
line: 13   char:  1   token: +
line: 13   char:  6   token: -
line: 13   char: 11   token: *
line: 13   char: 16   token: /
line: 16   char:  1   token: =
line: 16   char:  5   token: !=
line: 16   char: 10   token: <
line: 16   char: 14   token: <=
...
```

Interação Entre as Classes do Compilador e do Analisador Léxico



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.