Implementando um Analisador Léxico

ÉFREN L. SOUZA

Apresentação da Linguagem

DL (*Didactic Language*)

- DL é a linguagem que vamos implementar
- Possui características do C e do Pascal
- Comandos em português
- Os operadores aritméticos são os mesmos do C
- Os blocos possuem os delimitadores **`inicio`** e **`fim`**
- Os comandos de entrada e saída são palavras reservadas

DL

- Exemplo de um programa na linguagem DL

```
programa num_primo inicio
   inteiro n; inteiro x;
   booleano ehPrimo;
   leia(n);
   ehPrimo = verdadeiro;
   se (n <= 1) ehPrimo = falso;
   x = 2;
   enquanto( x <= n/2 & ehPrimo ) inicio
      se ( n % x == 0 )
         ehPrimo = falso;
      x = x + 1;
   fim
   se ( ehPrimo )
      escreva(1);
   senao
      escreva(0);
fim.
```

Tipos de Tokens

- Palavras Reservadas

- <PROGRAM, “programa”>
- <BEGIN, “inicio”>
- <END, “fim”>
- <INT, “inteiro”>
- <REAL, “real”>
- <BOOL, “booleano”>
- <TRUE, “verdadeiro”>
- <FALSE, “falso”>
- <READ, “leia”>
- <WRITE, “escreva”>
- <IF, “se”>
- <ELSE, “senao”>

Tipos de Tokens

- Atribuição

- <ASSIGN, “=”>

Tipos de Tokens

- Operadores Aritméticos

- <SUM, “+”>
- <SUB, “-”>
- <MUL, “*”>
- <DIV, “/”>

Tipos de Tokens

- Operadores Relacionais

- <EQ, "==">
- <NQ, "!=">
- <GT, ">">
- <LT, "<">
- <GE, ">=">
- <LE, "<=">

Tipos de Tokens

- Operadores Lógicos

- <LAND, “&”>
- <LOR, “|”>
- <LNOT, “!”>

Tipos de Tokens

- Outros símbolos

- <COMMA, “,”>
- <SEMI, “;”>
- <LPAREN, “(”>
- <RPAREN, “)”>

Tipos de Tokens

- Literais inteiros
- O lexema pode variar

- <LIT_INT, “16”>
- <LIT_INT, “256”>
- <LIT_INT, “1024”>

- Vamos usar o seguinte padrão para inteiros
  - DIGITO+

Tipos de Tokens

- Literais reais

- <LIT_REAL, “12.0”>
- <LIT_REAL, “0.55”>
- <LIT_REAL, “3.1415”>

- Vamos usar o seguinte padrão para reais
  - DIGITO+ . DIGITO*

Tipos de Tokens

- Identificadores

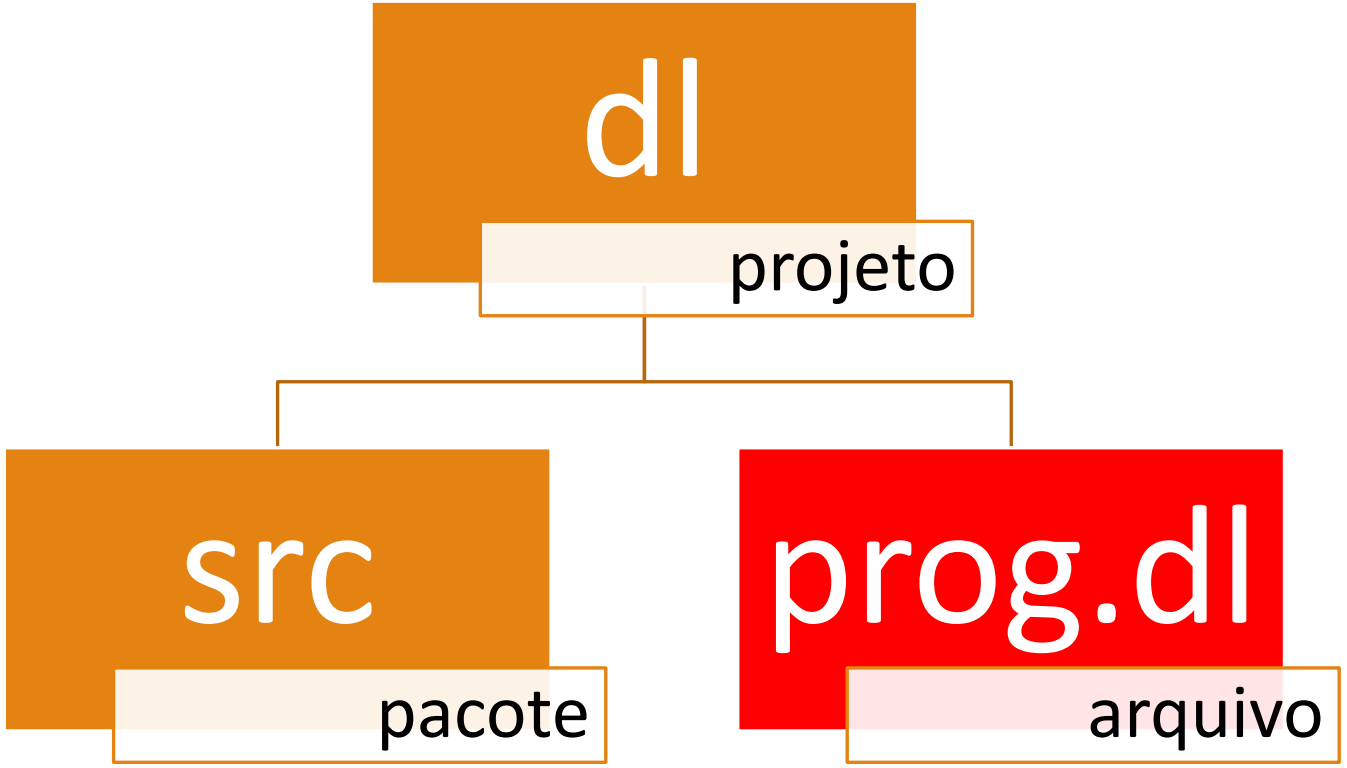
- <ID, “speed”>
- <ID, “_width”>
- <ID, “var30”>

- Vamos usar o seguinte padrão para identificadores
  - (LETRA | _) (LETRA | _ | DIGITO)*

Iniciando o Projeto

Projeto

- Inicie o Eclipse
- Crie um novo Projeto
  - File -> New -> Java Project
    - Nome: `dl`
    - Finish
- Na raiz, crie o arquivo vazio `prog.dl`



prog.dl

- Esse arquivo possui o código fonte que o analisador léxico irá ler

```
programa soma inicio
   inteiro a;
   inteiro b;
   inteiro c;
   a = 5;
   b = 3;
   c = a + b;
   escreva(c);
fim.
```

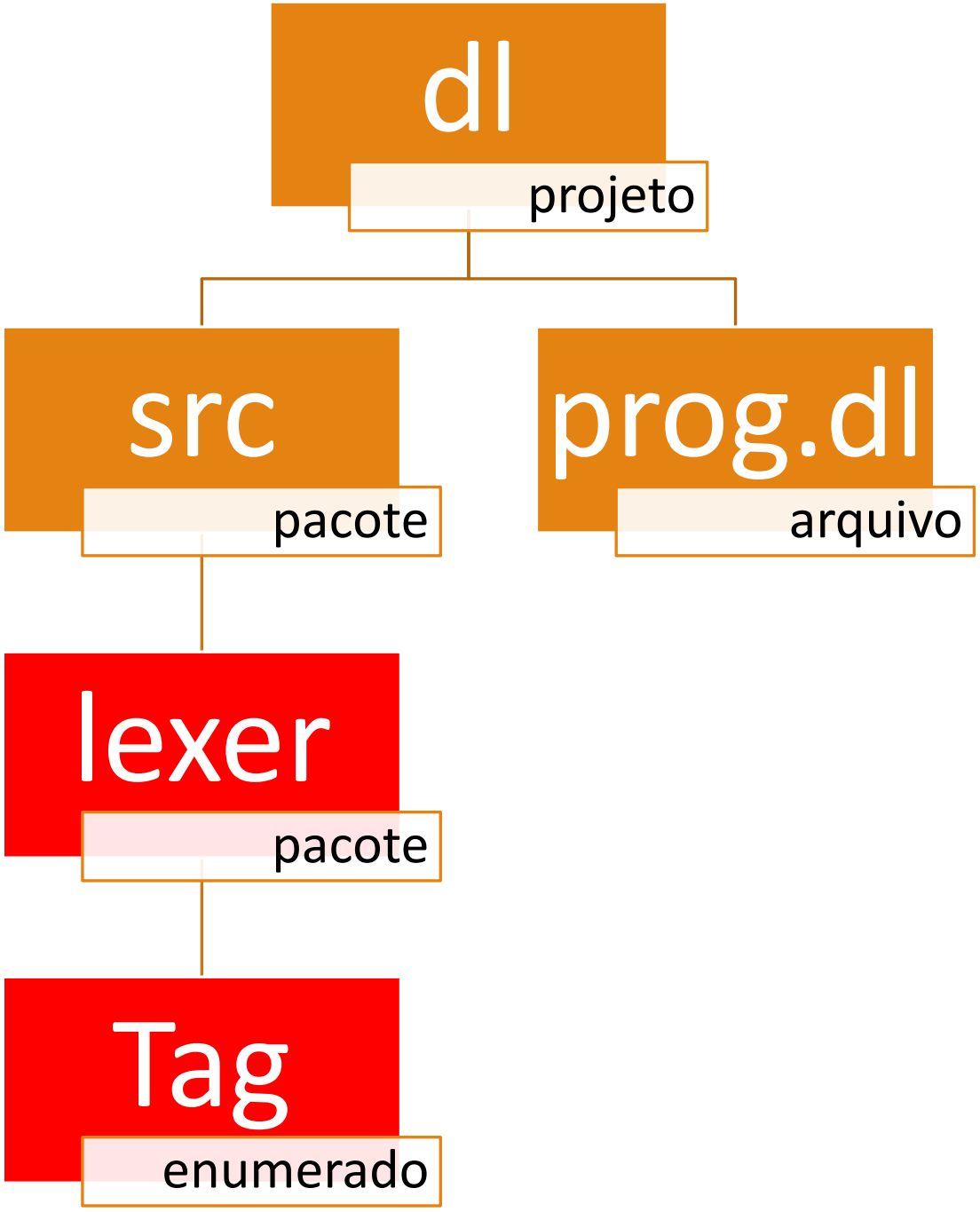
Classe Tag

Para que serve?

- A classe `Tag` é um tipo enumerado
- Ela serve para enumerar todos os tipos de *token* que o compilador irá gerar
- Essa enumeração será usada intensamente na fase de análise

Criando a Classe `Tag`

- Na pasta `src`, crie um novo pacote chamado `lexer`
  - Esse pacote conterá todas as classes do analisador léxico
- Nesse pacote crie um *enum* chamado `Tag`



Tag.java (i)

- Por enquanto, vamos enumerar uma pequena quantidade de tokens
- Cada novo tipo de *token* deve ser acrescentado nessa classe
- `EOF` é o *token* que marca o fim do código
- `UNK` é o *token* para padrões desconhecidos

```
public enum Tag {
    //Assign
    ASSIGN("ASSIGN"),
    //Arithmetical Operators
    SUM("SUM"), MUL("MUL"),
    //Logical Operators
    OR("OR"),
    //Relational Operators
    LT("LT"), LE("LE"), GT("GT"),
    //Others
    EOF("EOF"), UNK("UNK");
```

Tag.java (ii)

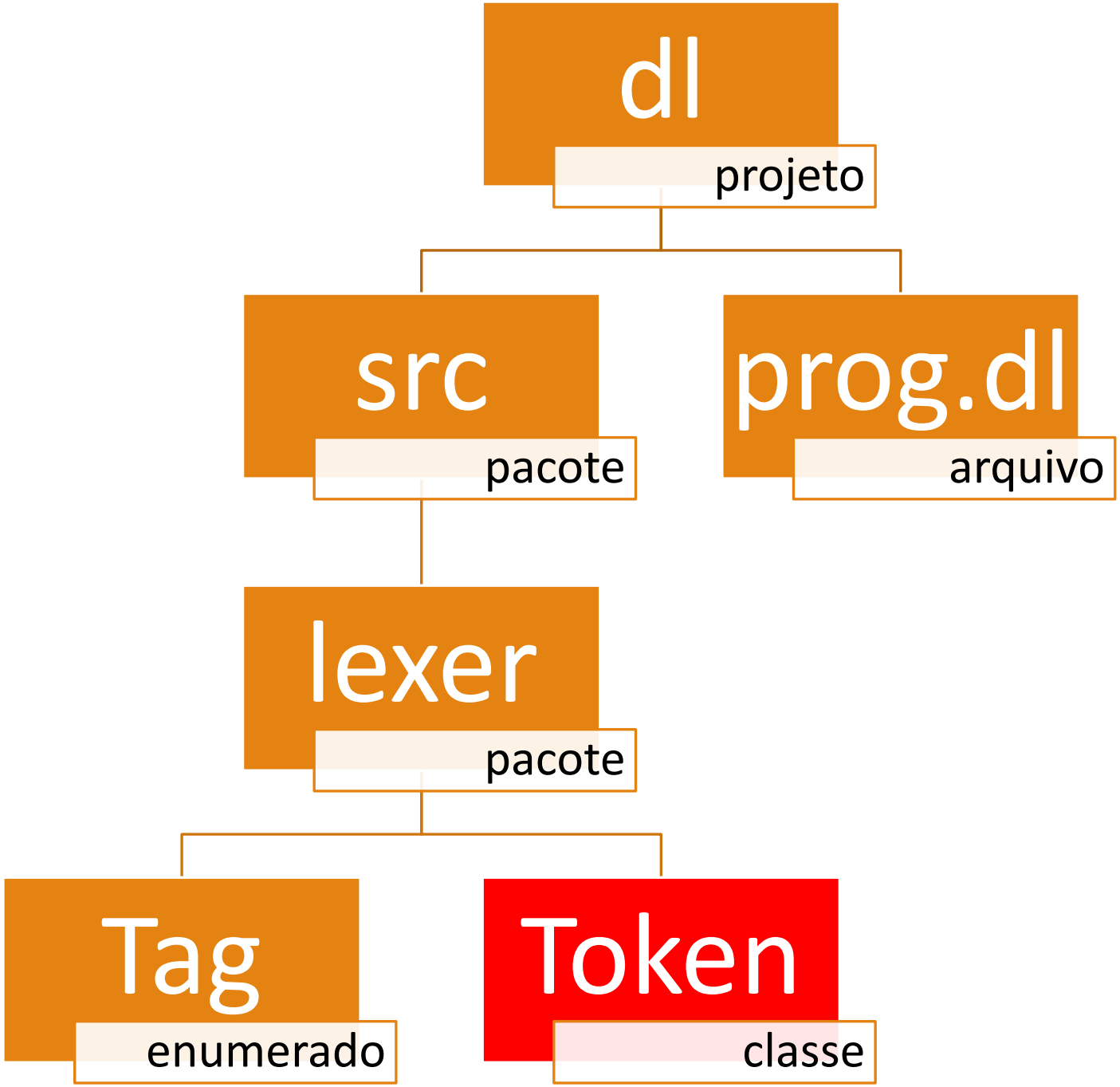
- Vamos adicionar um atributo ao tipo enumerado
- Esse novo atributo serve apenas para termos uma representação textual para cada elemento

```
    private String name;

    private Tag(String name) {
        this.name = name;
    }

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

Classe *Token*

Classe Token

- As instâncias dessa classe representam cada token identificado no código fonte da linguagem DL
- Na pasta `lexer` crie uma classe chamada `Token`



Token.java (i)

- O atributo `tag` guarda o tipo do *token*
- É possível criar um token apenas com tag
  - Apropriado para tokens que possuem sempre o mesmo lexema
  - Nesse caso, o lexema será nulo

```
public class Token {
    private Tag tag;
    private String lexeme;

    public Token(Tag t, String l) {
        tag = t;
        lexeme = l;
    }
```

Token.java (ii)

- Sobrescrever o método `toString`
- Serve para imprimir um *token* facilmente
  - $< LT >$
  - $< ID, 'max' >$

```
    public Tag tag() {
        return tag;
    }

    public String lexeme() {
        return lexeme;
    }

    @Override
    public String toString() {
        return "<" + tag +
                ", '" + lexeme + "'>";
    }
}
```

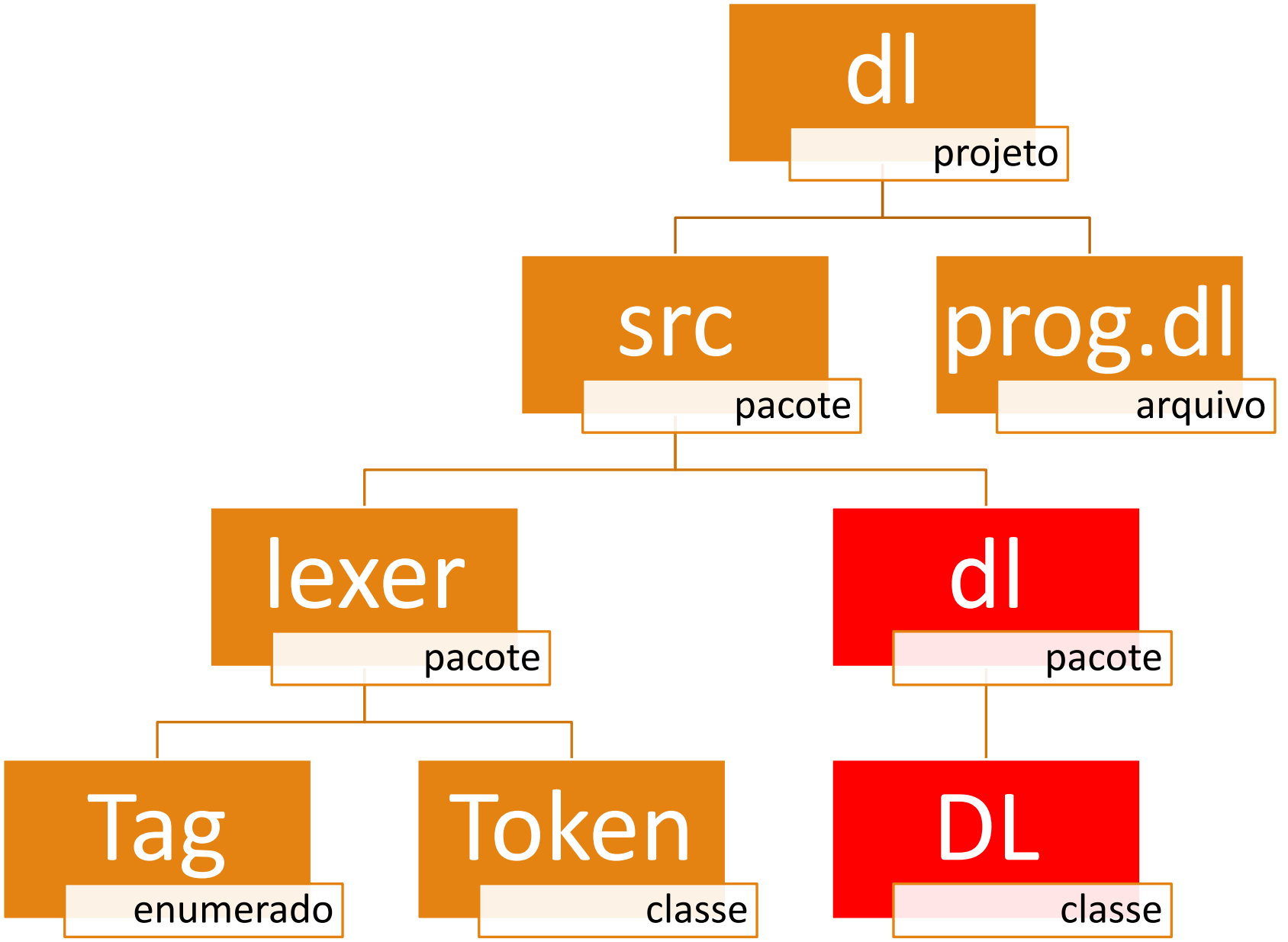
Classe DL

Para que serve?

- Classe principal do compilador
- É o ponto de partida para a execução do programa
- Agora vamos usá-la para executar o primeiro teste do nosso analisador léxico

Classe DL

- Crie o pacote `dl`
- Nesse pacote, crie a classe `DL`



DL.java

- Esse teste apenas cria 2 tokens e os imprime
- Um token possui lexema e o outro não
- Execute-o

```
public class DL {
    public static void main(String[] args) {
        Token t1 = new Token(Tag.ASSIGN, "=");
        Token t2 = new Token(Tag.LE, "<=");
        System.out.println(t1);
        System.out.println(t2);
    }
}
```

READING CODE right,

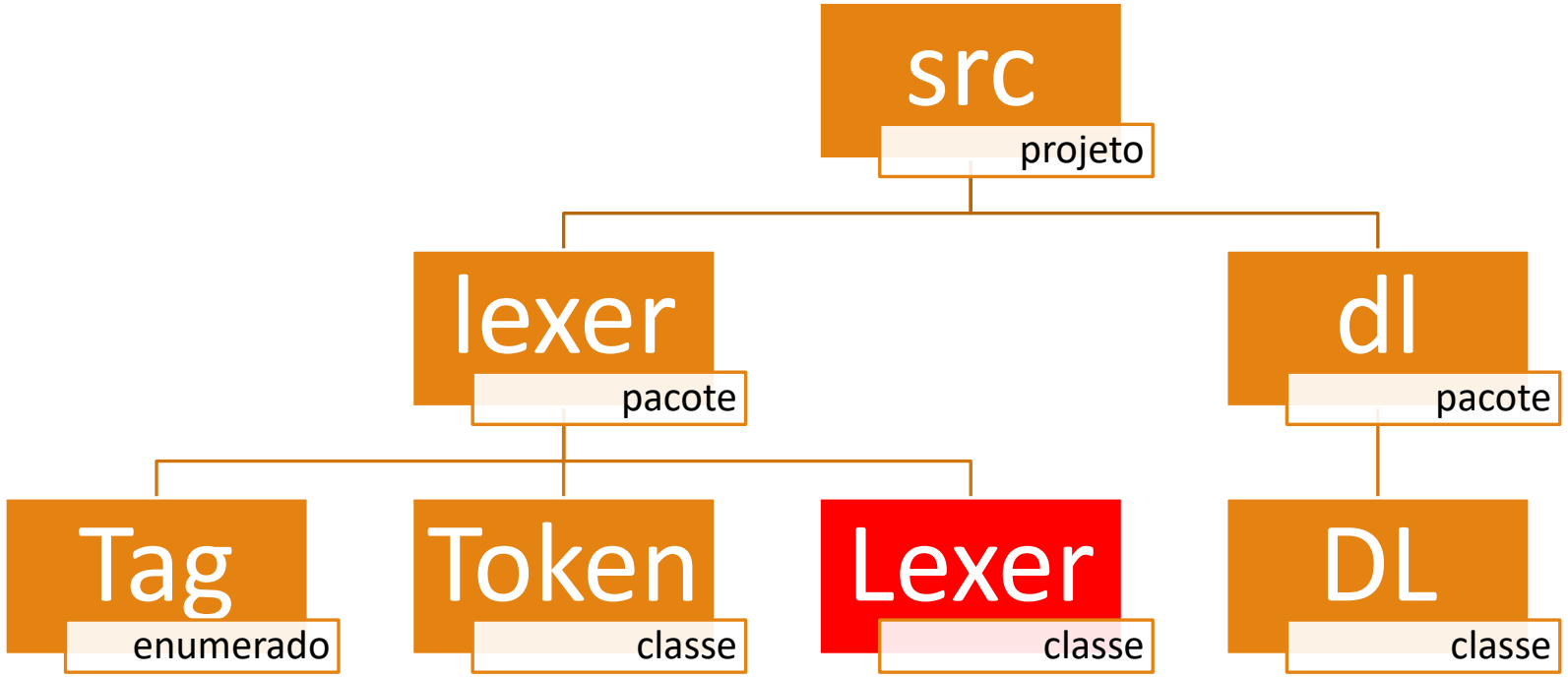
WITH SOME HELP FROM THE

Classe `Lexer`

Classe `Lexer`

- Essa classe representa o analisador léxico, essa deve ler todo o código fonte do programa que queremos extrair os *tokens*

- No pacote `lexer`, crie a classe `Lexer`



Lexer.java (atributos)

- `EOF_CHAR` é uma constante para o fim do arquivo
- O atributo `reader` representa o arquivo de entrada do compilador
- O atributo `line` é a linha do token que está sendo gerado
- O atributo `peek` é o último caractere lido

```
public class Lexer {
    private static final char
            EOF_CHAR = (char)-1;
    private static int line = 1;
    private BufferedReader reader;
    private char peek;
```

`Lexer.java` (construtor)

- Instancia o arquivo de entrada a partir de um parâmetro do construtor
- O primeiro caractere de `peek` é um espaço em branco

```
public Lexer(File file) {
    try {
        this.reader =
            new BufferedReader(
            new FileReader(file));
    } catch (Exception e) {
        e.printStackTrace();
    }
    this.peek = ' ';
}
```

Lexer.java (métodos)

- O método `nextChar()` atualiza o valor de `peek` para o próximo caractere do arquivo
- Além disso, se encontrar uma quebra de linha, incrementa o atributo `line`

```
public static int line() {
    return line;
}

private char nextChar() {
    if ( peek == '\n' ) line++;
    try {
        peek = (char)reader.read();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return peek;
}
```

Lexer.java (métodos)

- Esse método identifica espaços em branco para serem ignorados

```
private static boolean
        isWhitespace(int c) {
    switch (c) {
    case ' ': case '\t': case '\n':
        return true;
    default:
        return false;
    }
}
```

Lexer.java (identificando tokens)

- Ignora os espaços em branco
- Reconhece os lexemas de atribuição, soma e multiplicação

```
public Token nextToken() {
    while (isWhitespace(peek)) nextChar();
    switch(peek) {
    case '=':
        nextChar();
        return new Token(Tag.ASSIGN, "=");
    case '+':
        nextChar();
        return new Token(Tag.SUM, "+");
    case '-':
        nextChar();
        return new Token(Tag.SUB, "-");
    case '*':
        nextChar();
        return new Token(Tag.MUL,"*");
```

Lexer.java (identificando tokens)

- Reconhece os lexemas menor, menor ou igual, e maior
- Retorna um *token* para o fim do arquivo
- Retorna um *token* para qualquer lexema desconhecido

```
        case '|':
            nextChar();
            return new Token(Tag.OR, "|");
        case '<':
            nextChar();
            if ( peek == '=' ) {
                nextChar();
                return new Token(Tag.LE, "<=");
            }
            return new Token(Tag.LT, "<");
        case '>':
            nextChar();
            return new Token(Tag.GT, ">");
        case EOF_CHAR:
            return new Token(Tag.EOF, "");
        }
        String unk = String.valueOf(peek);
        nextChar();
        return new Token(Tag.UNK, unk);
    }
```

DL.java

- Vamos ajustar a classe DL para reconhecer os tokens
- Instancia a classe `Lexer` passando o arquivo com o código fonte como parâmetro
- **Faça o teste! Esse programa deve identificar alguns tokens, sendo muitos deles desconhecidos**

```
public class DL {
    public static void main(String[] args) {
        Lexer l =
            new Lexer(new File("prog.dl"));
        Token t = l.nextToken();
        while ( t.tag() != Tag.EOF ) {
            System.out.println(t);
            t = l.nextToken();
        }
    }
}
```

Exercício

Faça com que o programa agora reconheça os seguintes *tokens*

Tipo de Token	Lexema
NE, GT, GE, LT, LE	!= > >= < <=
LAND, LOR, LNOT	& | !
SUM, SUB, MUL, DIV	+ - * /
LPAREN, RPAREN	( )
COMMA, SEMI	, ;

Reconhecendo Literais

Literais Inteiros

- Os literais não são como os *tokens* anteriores

- Seus lexemas são dinâmicos, pois podem ser qualquer número inteiro

Tag.java

- Antes precisamos incluir o *tag* `LIT_INT` para ser usada nos tokens de literais inteiros

```
//Assign
ASSIGN("ASSIGN"),
//Arithmetical Operators
SUM("SUM"), MUL("MUL"),
//Logical Operators
OR("OR"),
//Relational Operators
LT("LT"), LE("LE"), GT("GT"),
//Literals
LIT_INT("LIT_INT"),
//Others
EOF("EOF"), UNK("UNK");
```

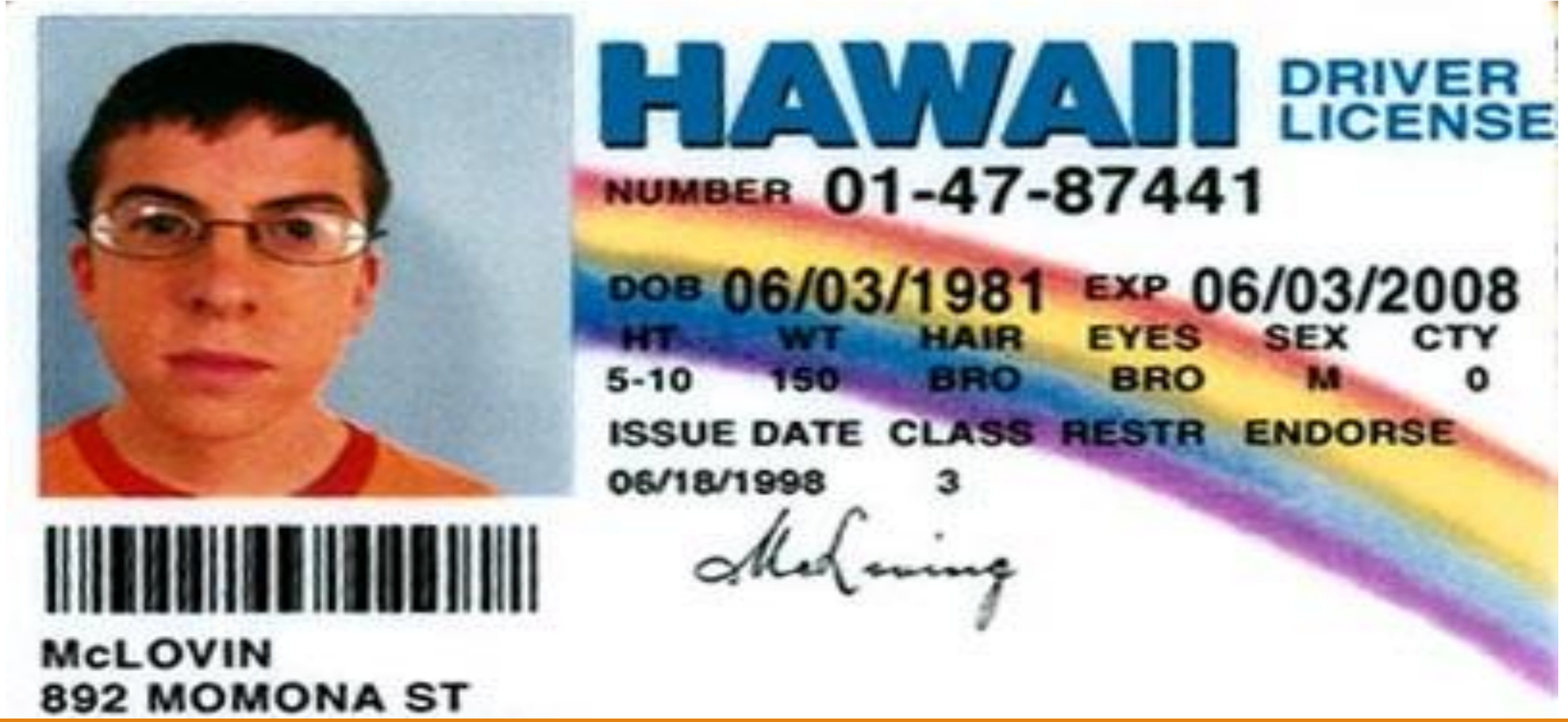
Lexer.java

- No método `nextToken()` adicione o trecho ao lado que reconhece o literal
- Coloque-o na seção *default* do *switch* (abaixo do EOF)
- Como ainda não estamos verificando os identificadores, esse código reconhecerá a parte numérica dos identificadores como números
- **Teste o código!**

```
case EOF_CHAR:
    return new Token(Tag.EOF);
default:
    if (Character.isDigit(peek)) {
        String num = "";
        do {
            num += peek;
            nextChar();
        } while( Character.isDigit(peek) );
        return new Token(Tag.LIT_INT, num);
    }
}
```

Exercício

- Faça com que o analisador léxico reconheça literais do tipo real
  - <LIT_REAL, “12.0”>
  - <LIT_REAL, “0.55”>
  - <LIT_REAL, “3.1415”>
  - <LIT_REAL, “15.”>
- Vamos usar o seguinte padrão para reais
  - DIGITO+ . DIGITO*



Reconhecendo Identificadores

Identificadores

- Os identificadores são usados para nomear variáveis, constantes, funções e etc

- Eles também são dinâmicos, pois podem incluir vários nomes

Tag.java

- Primeiramente devemos adicionar o *tag* `ID` ao conjunto de TAGs
- Um *token* de identificador não pode ser estático, pois seu lexema varia

```
//Assign
ASSIGN("ASSIGN"),
//Arithmetical Operators
SUM("SUM"), MUL("MUL"),
//Logical Operators
OR("OR"),
//Relational Operators
LT("LT"), LE("LE"), GT("GT"),
//Literals
LIT_INT("LIT_INT"),
//Identifiers
ID("ID"),
//Others
EOF("EOF"), UNK("UNK");
```

Lexer.java

- Vamos implementar um método que verifica se um caractere é aceito como o início de um identificador (não pode ser um número)
- E outro método que verifica se um caractere é parte de um identificador

```
private static boolean
  isIdStart(int c) {
    return
      (Character.isAlphabetic(c)
            || c == '_' );
}

private static boolean
  isIdPart(int c) {
    return
      (isIdStart(c)
        || Character.isDigit(c));
}
```

Lexer.java

- Adicionar o fragmento de código no *switch* do método *nextToken()*
- Coloque-o após a condição que verifica o LIT_INT
- Por enquanto, as palavras reservadas serão reconhecidas como identificadores
- **Faça o teste!**

```
} else if ( isIdStart(peek)   ) {
    String id = "";
    do {
        id += peek;
        nextChar();
    } while ( isIdPart(peek) );
    return new Token(Tag.ID, id);
}
```

Reconhecendo Palavras reservadas

Palavras Reservadas

- As palavras reservadas se encaixam no padrão dos identificadores

- Para fazermos a diferenciação, precisamos registrar as palavras reservadas em uma tabela para posterior verificação

Tag.java

- Primeiramente vamos definir os *tags* das palavras reservadas

```
//Reserved Words
PROGRAM("PROGRAM"),
BEGIN("BEGIN"), END("END"),
//Assign
ASSIGN("ASSIGN"),
//Arithmetical Operators
SUM("SUM"), MUL("MUL"),
//Logical Operators
OR("OR"),
//Relational Operators
LT("LT"), LE("LE"), GT("GT"),
//Literals
LIT_INT("LIT_INT"),
//Identifiers
ID("ID"),
//Others
EOF("EOF"), UNK("UNK");
```

Lexer.java

- Para diferenciar as palavras reservadas, essas são colocadas em uma tabela
- Um *hash* pode ser usada para isso
- A tabela `keywords` será usada para registrar as palavras reservadas

```
public class Lexer {
    private static final char
    EOF_CHAR = (char)-1;
    private static int line = 1;
    private BufferedReader reader;
    private char peek;
    private Hashtable<String, Tag>
        keywords;
```

Lexer.java

- O método privado `reserve` adiciona à tabela um *token* que corresponde a uma palavra reservada
- Esse método é chamado no construtor uma vez para cada palavra chave

```
public Lexer(File file) {
    try {
        this.reader =
            new BufferedReader(
            new FileReader(file));
    } catch (Exception e) {
        e.printStackTrace();
    }
    this.peek = ' ';
    keywords = new Hashtable<String, Tag>();
    keywords.put("programa", Tag.PROGRAM);
    keywords.put("inicio", Tag.BEGIN);
    keywords.put("fim", Tag.END);
}
```

Lexer.java

- Agora antes de gerarmos um identificador, devemos verificar antes se não se trata de uma palavra reservada
- Se for uma palavra reservada, o próprio *token* registrado na tabela é retornado

```
} else if ( isIdStart(peek)   ) {
    String id = "";
    do {
        id += peek;
        nextChar();
    } while ( isIdPart(peek) );
    if ( keywords.containsKey(id) )
        return new Token(
                keywords.get(id), id);
    return new Token(Tag.ID, id);
}
```

Exercício

- Faça com que o lexer identifique as palavras reservadas da tabela

Tipo de Token	Lexema
PROGRAM	programa
BEGIN	inicio
END	fim
INT	inteiro
REAL	real
BOOL	booleano
TRUE	verdadeiro
FALSE	falso
READ	leia
WRITE	escreva

Exercício

- Faça com que nosso analisador léxico ignore comentários de uma linha da mesma forma que os comentários do C

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.