

JAVACC

Carolina Evaristo Scarton
carol@icmc.usp.br

O que é?

- JAVACC – JAVA Compiler Compiler - é um gerador de analisadores léxicos e de *parsers*
 - Analisador léxico: quebra uma sentença em tokens e classifica estes tokens
 - *Parser*: analisa uma sentença de acordo com as regras da gramática
- COMPILADOR / INTERPRETADOR → incorpora um analisador léxico e um *parser*
 - O analisador semântico e a geração de código são possíveis via inclusão de código JAVA

O que é?

- O JAVACC:
 - Lê uma gramática no formato **EBNF**
 - **gramática LL(k)**
 - Converte a gramática para um programa em JAVA que pode **reconhecer programas** para a dada gramática
 - A descrição dos tokens é dada em **expressão regular**
 - está no mesmo arquivo da gramática

3

O que é?

- A **entrada** do **analisador léxico** é uma sequência de caracteres
- A **saída** do **analisador léxico** é dada pelo JAVACC → uma sequência de objetos da classe Token
- A **entrada** do **analisador sintático** é dada pelo JAVACC → **saída** do **analisador léxico**
- A **saída** do **analisador sintático** não é predefinida pelo JAVACC → definida pelo programador
 - Deve estar em JAVA

4

O que é?

- Pode gerar uma representação intermediária na forma de uma árvore sintática abstrata via ferramenta JJTree, que acompanha o JavaCC
- Pode gerar um documento HTML com a especificação dos tokens e da gramática via ferramenta JJDoc

5

Instalação

- Instale o JAVA (jdk 6.0, por exemplo)
- Faça o download do JAVACC e descompacte:
<http://java.net/projects/javacc/downloads/download/javacc-5.0.tar.gz>
- Adicione o diretório /bin do JAVACC na variável de ambiente do sistema PATH

6

LL(k)

- L - **Left-right**: sentido de leitura $\rightarrow$ da esquerda para a direita
- L – **Leftmost derivation**: tipo de derivação considerada $\rightarrow$ derivação mais à esquerda
- (k) – **k lookahead**: número de símbolos necessários para distinguir a produção correta

7

LL(k)

- **LL(1)**:
 - Dada: $A \rightarrow X_1A_1 \mid X_2A_2 \mid \dots \mid X_nA_n \rightarrow \text{First}(X_i)$ são disjuntos 2 a dois.
 - Para toda produção $A \rightarrow \alpha \mid \beta$:
 - Se $\beta \Rightarrow^* \lambda \rightarrow \alpha$ não deriva cadeias começando com um terminal no $\text{Follow}(A)$
 - $\text{First}(\alpha) \neq \text{Follow}(A)$
 - Se $\alpha \Rightarrow^* \lambda \rightarrow \beta$ não deriva cadeias começando com um terminal no $\text{Follow}(A)$
 - $\text{First}(\beta) \neq \text{Follow}(A)$

8

Algumas restrições - LL(k)

- A gramática não pode conter recursão a esquerda:

```
E -> T | E "+" T
T -> F | T "*" F
F -> NUM | "(" E ")"
```

ERRADO

```
Java Compiler Compiler Version 5.0 (Parser Generator)
<type "javacc" with no arguments for help>
Reading from file ex1.jj . . .
Error: Line 34, Column 1: Left recursion detected: "h... --> h..."
Error: Line 27, Column 1: Left recursion detected: "Start... --> Start..."
Detected 2 errors and 0 warnings.
```

```
E -> T ("+" T)*
T -> F ("*" F)*
F -> NUM | "(" E ")"
```

CORRETO

```
Java Compiler Compiler Version 5.0 (Parser Generator)
<type "javacc" with no arguments for help>
Reading from file ex1.jj . . .
File "TokenMgrError.java" does not exist. Will create one.
File "ParseException.java" does not exist. Will create one.
File "Token.java" does not exist. Will create one.
File "SimpleCharStream.java" does not exist. Will create one.
Parser generated successfully.
```

Exemplos retirados de: Using JAVACC: <http://www.cs.lmu.edu/~ray/notes/javacc/>

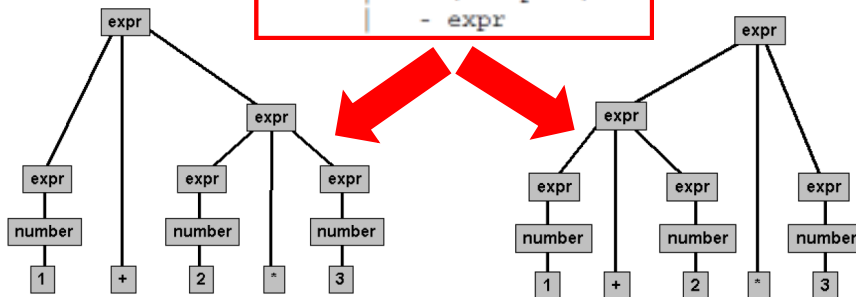
9

Algumas restrições - LL(k)

- A gramática não pode ser ambígua:

```
expr    :=  number
         |  expr '+' expr
         |  expr '-' expr
         |  expr '*' expr
         |  expr '/' expr
         |  '(' expr ')'
```

**Gera duas árvores
sintáticas para a
expressão 1+2*3**

Exemplos retirados de: Building your own languages with JAVACC: <http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-coolttools.html>

10

Algumas restrições - LL(k)

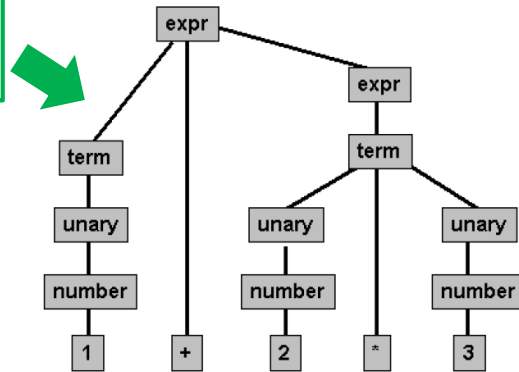
- A gramática não pode ser ambígua:

```

expr -> term() ( "+" expr | "-" expr ) *
term -> unary() ( "*" term | "/" term ) *
unary -> "-" element | element
element -> <NUMBER> | "(" expr ")"

```

Gera somente uma
árvores sintática
para 1+2*3



Exemplos retirados de: Building your own languages with JAVACC: <http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-coolttools.html>

11

LL(1)

- **Observação importante:**
 - Se a gramática for **LL(1)** sem **recursão a esquerda** → a gramática **não é ambígua!**
- Exercício:
 - Verificar se a Frankie é LL(1)

12

LL(1)

- Exercício:
 - Verificar se a Frankie é LL(1)

Frankie não é LL(1)!!!

13

Primeiro exemplo - Calculadora

- Realizar operações de soma, subtração e multiplicação com número inteiros
- Exemplo de funcionamento:
 - Operação = $1+2 \rightarrow \text{Resultado} = 3$
 - Operação = $-2+1 \rightarrow \text{Resultado} = -1$
- Gramática:

Start $\rightarrow$ Expression
 Expression $\rightarrow$ Term (PLUS Term | MINUS Term)*
 Term $\rightarrow$ Primary (TIMES Primary)*
 Primary $\rightarrow$ <NUMBER>

14

Primeiro exemplo - Calculadora

```

PARSER BEGIN(Calculator)
import java.io.PrintStream ;
class Calculator {
    static void main( String[] args )
    throws ParseException, TokenMgrError {
        Calculator parser = new Calculator(System.in);
    };
    parser.Start(System.out);
}
double previousValue = 0.0 ;
}
PARSER END(Calculator)

```

A classe principal é "Calculator"

Classes de exceções geradas pelo JAVACC

Entrada dos tokens

Método gerado pelo símbolo inicial da gramática

Exemplo adaptado de: JAVACC Tutorial: <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/>

15

Primeiro exemplo - Calculadora

SKIP : { " " | "\t" } Caracteres desconsiderados durante a análise

Tokens são definidos por ERs

```

TOKEN : { < EOL : "\n" | "\r" | "\r\n" > }
TOKEN : { < PLUS : "+" > }
TOKEN : { < MINUS : "-" > }
TOKEN : { < TIMES : "*" > }
TOKEN : { < NUMBER : <DIGITS> > }
TOKEN : { < #DIGITS : ([ "0" - "9" ] ) + > }

```

Token local

O JAVACC trata EOF automaticamente

Exemplo adaptado de: JAVACC Tutorial: <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/>

16

Primeiro exemplo - Calculadora

```
void Start(PrintStream printStream):
{
    (
        previousValue = Expression()
        <EOL>
        { printStream.println( previousValue
        ); }
    )*
    <EOF>
}
```

**Cada não
terminal da
gramática é
um método
do parser**

Código JAVA inserido

```
int Primary() :
{
    Token t ;
}
{
    t = <NUMBER>
    try{
        return Integer.parseInt( t.image );
    }catch (NumberFormatException e){
        System.out.println("\nErro Semantico na linha " +
            t.beginLine + ", coluna " + t.beginColumn + ": overflow");
    }
}
```

Exemplo adaptado de: JAVACC Tutorial: <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/>

17

Primeiro exemplo - Calculadora

```
int Term():
{
    int i ;
    int value ;
}
{
    value = Primary()
    (
        <TIMES>
        i = Primary()
        { value *= i ; }
    )*
    { return value ; }
}
```

**Cada não
terminal da
gramática é
um método
do parser**

Código JAVA inserido

```
int Expression():
{
    int i ;
    int value ;
}
{
    value = Primary()
    (
        <PLUS>
        i = Primary()
        { value += i ; }
        |
        <MINUS>
        i = Primary()
        { value -= i ; }
    )*
    { return value ; }
}
```

Exemplo adaptado de: JAVACC Tutorial: <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/>

18

Primeiro exemplo - Calculadora

- Gerando o parser:
 - javacc calculator.jj → Gera 7 arquivos:
 - Calculator.java
 - CalculatorConstants.java
 - CalculatorTokenManager.java
 - ParseException.java
 - SimpleCharStream.java
 - Token.java
 - TokenMgrError.java
 - javac Calculator.java
 - java Calculator

19

Primeiro exemplo - Calculadora

- Calculator.java → parser
- CalculatorTokenManager → analisador léxico
- TokenMgrError → para detectar erros léxicos
 - Subclasse de Throwable
- ParseException → para dectar erros sintáticos
 - Subclasse de Exception (que é subclasse de Throwable)
- Token → classe que representa os tokens
 - image → string com o token
 - kind → número que representa o tipo do token
- SimpleCharStream → responsável por adaptar a entrada para o analisador léxico

20

Primeiro exemplo - Calculadora

- Alguns testes:
 - Operação: $23 * 80 \rightarrow$ Resultado: 1840
 - Operação: $-55 + 33 \rightarrow$ Resultado: -22
 - Operação: $-55 + -10 \rightarrow$ Resultado: -65
 - Operação: $100 * -1 \rightarrow$ Resultado: -100

```
D:\Documentos\usp\PfE>java Calculator
23*80
1840
-55+33
-22
-55+-10
-65
100*-1
-100
```

21

Primeiro exemplo - Calculadora

- Mensagens para erros léxicos podem ser editadas no arquivo **TokenMgrError.java**
- Exemplo de erro léxico $\rightarrow$ uso do operador “/” não definido
 - Operação: $33/1 \rightarrow$ Erro léxico!

```
D:\Documentos\usp\PfE>java Calculator
33/1
Erro lexico encontrado na linha 1, coluna 3. Encontrado: "/" <47>, depois de :
```

22

Primeiro exemplo - Calculadora

- Mensagens para erros sintáticos podem ser editadas no arquivo **ParserException.java**
- Exemplo de erro sintático → algo não definido nas regras da gramática
 - Operação: +33 → Erro sintático!

```
D:\Documentos\usp\P&E>java Calculator
+33
Erro sintatico. Encontrado " "+" " na linha 1, coluna 1.
era esperado:
<EOF>
<NUMBER> ...
```

23

Primeiro exemplo - Calculadora

- Erros semânticos são identificados durante a análise sintática
- Exemplo de erro semântico → overflow
 - Operação: 777766669999+33 → Erro semântico!

```
try{
    return Integer.parseInt( t.image );
}catch (NumberFormatException e){
    System.out.println("\nErro Semantico na linha " +
        t.beginLine + ", coluna " + t.beginColumn + ":
        overflow"); }
```

Trecho de código retirado do método Primary() → identificação de erro semântico

```
D:\Documentos\usp\P&E>java Calculator
777766669999+33
Erro Semantico na linha 1, coluna 1: overflow
```

24

Primeiro exemplo - Calculadora

- Exercício:
 - Incluir operações de negação e operações parentizadas na calculadora

25

Primeiro exemplo - Calculadora

- Exercício:
 - Incluir operações de negação e operações parentizadas na calculadora

```
Start -> Expression  
Expression -> Term (PLUS Term | MINUS Term)*  
Term -> Primary (TIMES Primary)*  
Primary -> <NUMBER>  
| <OPEN_PAR> Expression <CLOSE_PAR>  
| <MINUS> Primary
```

26

Primeiro exemplo - Calculadora

- Exercício:
 - Incluir operações de negação e operações parentizadas na calculadora

```
int Primary() :
{
    Token t ;
    Int d;
}
{
    t = <NUMBER>
    {
        try{
            return Integer.parseInt( t.image );
        }catch (NumberFormatException e){
            System.out.println("\nErro Semantico na linha " +
                t.beginLine + ", coluna " + t.beginColumn + ": overflow");
        }
    }

    <OPEN_PAR> d=Expression() <CLOSE_PAR>
    { return d ; }

    <MINUS> d=Primary()
    { return -d ; }
}
```

TOKEN : { < OPEN_PAR : "(" > }
 TOKEN : { < CLOSE_PAR : ")" > }

27

Regras de Desambiguação

- Como o JAVACC realiza a desambiguação
- Exemplo → expressão regular:
 - Palavra reservada: *program*
 - Identificador: *programa*
- Duas regras:
 - Busca o maior prefixo que define um token válido
 - Se uma string corresponde a dois tipos → usar o tipo que é definido primeiro no arquivo

28

Lookahead

- No exemplo da calculadora a gramática é LL(1)
 - O compilador só precisa do próximo token para decidir
- **E se LL(k) com $k > 1$?**

```
comando -> id ":" expressao
comando -> id "(" expressao (, expressao)* ")"
```

29

Lookahead

- No exemplo da calculadora a gramática é LL(1)
 - O compilador só precisa do próximo token para decidir
- **E se LL(k) com $k > 1$?**

Duas soluções

```
comando -> id ":" expressao
comando -> id "(" expressao (, expressao)* ")"
```

30

Lookahead

- E se LL(k) com $k > 1$?

- 1ª solução: Transformar a gramática em LL(1)

```
comando -> <ID> ":" expressao
comando -> <ID> "(" expressao (, expressao)* ")"
```

```
comando -> <ID> atrib_decl
Atrib_decl -> ":" expressao | "(" expressao (, expressao)* ")"
```

31

Lookahead

- E se LL(k) com $k > 1$?

- 2ª solução: Usar a opção LOOKAHEAD (2)

```
comando -> <ID> ":" expressao
comando -> <ID> "(" expressao (, expressao)* ")"
```

```
void comando():
{
  { LOOKAHEAD(2) <ID> ":" expressao | <ID> "("
    expressao (, expressao)* ")" }
```

32

Projeto → Parte 1

- **Analizador Léxico:**

- Criar um analisador léxico que tabula a saída código/token para FRANKIE com as extensões do grupo
- Exemplo → analisador léxico para PS

33

Projeto → Parte 1

- `<programa> ::= program <identificador> ; <bloco>.`
- `<bloco> ::=`
`[<parte de declarações de variáveis>]`
`[<parte de declarações de procedimentos>]`
`<comando composto>`
- `<tipo> ::= <identificador>`
- `<parte de declarações de variáveis> ::=`
`var <declaração de variáveis>`
`{; <declaração de variáveis>;}`
- `<declaração de variáveis> ::=`
`<lista de identificadores> : <tipo>`

34

Projeto → Parte 1

- `<lista de identificadores> ::=`
`<identificador> {, <identificador>}`
- `<parte de declarações de procedimentos> ::=`
`{declaração de procedimento> ;}`
- `<declaração de procedimento> ::=`
`procedure <identificador>`
`[<parâmetros formais>] ; <bloco>`
- `<parâmetros formais> ::=`
`( <seção de parâmetros formais>`
`{ ; <seção de parâmetros formais> } )`
- `<seção de parâmetros formais> ::=`
`[var] <lista de identificadores> : <identificador>`

35

Projeto → Parte 1

- `<comando composto> ::= begin <comando> { ; <comando> } end`
- `<comando> ::=`
`<atribuição>`
`| <chamada de procedimento>`
`| <comando composto>`
`| <comando condicional 1>`
`| <comando repetitivo 1>`
- `<atribuição> ::= <variável> := <expressão>`
- `<chamada de procedimento> ::=`
`<identificador> [ ( <lista de expressões> ) ]`
`<comando condicional 1> ::=`
`if <expressão> then <comando>`
`[else <comando>]`
`<comando repetitivo 1> ::=`
`while <expressão> do <comando>`

36

Projeto → Parte 1

- `<expressão> ::=`
`<expressão simples> [<relação> <expressão simples>]`
- `<relação> ::=` `=` | `<>` | `<` | `<=` | `>=` | `>`
- `<expressão simples> ::=`
`[+ | -] <termo> {(+ | - | or) <termo>}`
- `<termo> ::=`
`<fator> {( * | div | and ) <fator> }`
- `<fator> ::=`
`<variável>`
`| <número>`
`| ( <expressão> )`
`| not <fator>`
- `<variável> ::=` `<identificador>`
- `<lista de expressões> ::=` `<expressão> { , <expressão> }`

37

Projeto → Parte 1

```

PARSER_BEGIN (AnaLex)
    public class AnaLex{
        public static void main(String[] args){
            AnaLex parser = new AnaLex(System.in);
            while(true){
                try {
                    parser.parser();
                } catch (ParseException ex) {
                    System.out.println(ex.getMessage());
                    System.exit(-1);
                } catch (TokenMgrError ex) {
                    System.out.println(ex.getMessage());
                    System.exit(-1);
                }
            }
        }
    }
PARSER_END (AnaLex)

```

38

Projeto → Parte 1

```

/**
 * Eliminando comentarios e espacos
 */
SKIP: { " " | "\r" | "\t" | "\n" }
SKIP: { <BLOCKCOMMENT: "{" (~["}"])* ">" }
SKIP: { <LINECOMMENT: "//" (~["\n"])* (" \n" | "\r" | "\r\n")> }

/**
 * Palavras reservados
 */
TOKEN: { <AND: "and"> }
TOKEN: { <BEGIN: "begin"> }
TOKEN: { <DIV: "div"> }
TOKEN: { <DO: "do"> }
TOKEN: { <ELSE: "else"> }
TOKEN: { <END: "end"> }
TOKEN: { <IF: "if"> }
TOKEN: { <NOT: "not"> }
TOKEN: { <OR: "or"> }
TOKEN: { <PROCEDURE: "procedure"> }
TOKEN: { <PROGRAM: "program"> }
TOKEN: { <THEN: "then"> }
TOKEN: { <VAR: "var"> }
TOKEN: { <WHILE: "while"> }

```

39

Projeto → Parte 1

```

/**
 * Simbolos
 */
TOKEN: { <GREATER: ">" }
TOKEN: { <GREATEREQ: ">=" }
TOKEN: { <NOTEQUAL: "<" }
TOKEN: { <EQUAL: "=" }
TOKEN: { <LESS: ">" }
TOKEN: { <LESSEQ: ">=" }
TOKEN: { <MINUS: "-" }
TOKEN: { <COMMA: "," }
TOKEN: { <SEMICOLON: ";" }
TOKEN: { <COLON: ":" }
TOKEN: { <ASSIGNMENT: ":=" }
TOKEN: { <DOT: "." }
TOKEN: { <LPAREN: "(" }
TOKEN: { <RPAREN: ")" }
TOKEN: { <STAR: "*" }
TOKEN: { <PLUS: "+" }

```

40

Projeto → Parte 1

```

/**
 * Tokens auxiliares
 */
TOKEN: { <#LETTER: ["a"-"z"] | ["A"-"Z"]> }
TOKEN: { <#DIGIT: ["0"-"9"]> }
TOKEN: { <#UNDERLINE: "_"> }

/**
 * Inteiros
 */
TOKEN: { <NUMBER: (<DIGIT>)+> }

/**
 * Identificadores
 */
TOKEN: { <ID: (<LETTER> | <UNDERLINE>) (<LETTER> | <DIGIT> | <UNDERLINE>)* > }

```

41

Projeto → Parte 1

```

void parser() throws TokenMgrError, ParseException :
{
    Token t;

    "+t.toString() ); } t=<AND> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tAND
    "+t.toString() ); } t=<BEGIN> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tBEGIN
    "+t.toString() ); } t=<DIV> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tDIV
    "+t.toString() ); } t=<DO> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tDO
    "+t.toString() ); } t=<ELSE> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tELSE
    "+t.toString() ); } t=<END> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tEND
    "+t.toString() ); } t=<IF> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tIF
    "+t.toString() ); } t=<NOT> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tNOT
    "+t.toString() ); } t=<OR> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tOR
    "+t.toString() ); } t=<PROCEDURE> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tPROCEDURE -
    "+t.toString() ); } t=<PROGRAM> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tPROGRAM -
    "+t.toString() ); } t=<THEN> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tTHEN
    "+t.toString() ); } t=<VAR> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tVAR
    "+t.toString() ); } t=<WHILE> { System.out.println("@(" +t.beginLine+", "+t.beginColumn+")\tWHILE

```

42

Projeto → Parte 1

```

/**
 * Símbolos
 */
" + t.toString() ); } t=<GREATER> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t GREATER -
" + t.toString() ); } t=<GREATEREQ> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t GREATEREQ -
" + t.toString() ); } t=<NOTEQUAL> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t NOTEQUAL -
" + t.toString() ); } t=<EQUAL> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t EQUAL -
" + t.toString() ); } t=<LESS> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t LESS -
" + t.toString() ); } t=<LESSEQ> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t LESSEQ -
" + t.toString() ); } t=<MINUS> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t MINUS -
" + t.toString() ); } t=<COMMA> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t COMMA -
" + t.toString() ); } t=<SEMICOLON> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t SEMICOLON -
" + t.toString() ); } t=<COLON> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t COLON -
" + t.toString() ); } t=<ASSIGNMENT> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t ASSIGNMENT -
" + t.toString() ); } t=<DOT> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t DOT -
" + t.toString() ); } t=<LPAREN> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t LPAREN -
" + t.toString() ); } t=<RPAREN> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t RPAREN -
" + t.toString() ); } t=<STAR> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t STAR -
" + t.toString() ); } t=<PLUS> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t PLUS -

```

43

Projeto → Parte 1

```

/**
 * Inteiros
 */
" + t.toString() ); } t=<NUMBER> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t NUMBER
/**
 * Identificadores
 */
" + t.toString() ); } t=<ID> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ") \t ID
} | <EOF> { System.exit(0); }

```

Há outra maneira de fazer a tabulação?

44

Projeto → Parte 1

```

/**
 * Inteiros
 */
t=<NUMBER> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ")\" + t.toString() ); }
/**
 * Identificadores
 */
t=<ID> { System.out.println("@(" + t.beginLine + ", " + t.beginColumn + ")\" + t.toString() ); }
<EOF> { System.exit(0); }

```

Talvez usando a opção
DEBUG_PARSER = true

Há outra maneira

45

Projeto → Parte 1

```

program teste1;
var x: integer;
begin
    x := 3;
    if x > 5 then
    begin
        x := 5;
    end else
    begin
        x := 0;
    end
end.

```

46

SCCo206 – Introdução à Compilação – JAVACC – 14/03/2011

Projeto → Parte 1

Saída do
Analizador
Léxico



```
@(1,1) PROGRAM - program
@ (1,9) ID - testel
@ (1,15) SEMICOLON - ;
@ (2,1) VAR - var
@ (2,5) ID - x
@ (2,6) COLON - :
@ (2,8) ID - integer
@ (2,15) SEMICOLON - ;
@ (3,1) BEGIN - begin
@ (4,9) ID - x
@ (4,11) ASSIGNMENT - :=
@ (4,14) NUMBER - 3
@ (4,15) SEMICOLON - ;
@ (5,9) IF - if
@ (5,12) ID - x
@ (5,14) LESS - >
@ (5,16) NUMBER - 5
@ (5,18) THEN - then
@ (6,6) BEGIN - begin
@ (7,17) ID - x
@ (7,19) ASSIGNMENT - :=
@ (7,22) NUMBER - 5
@ (8,9) END - end
@ (8,13) ELSE - else
@ (9,6) BEGIN - begin
@ (10,17) ID - x
@ (10,19) ASSIGNMENT - :=
@ (10,22) NUMBER - 0
@ (11,9) END - end
@ (12,1) END - end
@ (12,4) DOT - .
```

47

SCCo206 – Introdução à Compilação – JAVACC – 14/03/2011

Projeto → Parte 1

- Erros devem ser adaptados
 - Para analisador léxico → TokenMgrError.java

48

Referências

- **Building your own languages with JAVACC:**
<http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html>
- **JAVACC Tutorial:** <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/>
- **Using JAVACC:** <http://www.cs.lmu.edu/~ray/notes/javacc/>
- **Java Compiler Compiler Documentation:** <http://javacc.java.net/doc/>
- **JAVACC: Java compiler's compiler:**
<http://litiwww.epfl.ch/~petitp/GenieLogiciel/GenLog7.pdf>
- **Writing Interpreters with JAVACC:**
<http://www.cs.nmsu.edu/~rth/cs/cs471/InterpretersJavaCC.html>
- **A Start Kit for JAVACC:** <http://w3.msi.vxu.se/users/jonasl/javacc/>
- **Gramáticas LL(k) – Notas de Aula:**
http://winandy.voila.net/LLk_NotasAula.pdf
- **Notas de aula da professora Sandra Maria Aluísio**
- **Notas de aula do professor Thiago Pardo**
- Aho, A. V., Lam, M. S., Sethi, R. e Ullman, J. D. (2008): **Compiladores: Princípios, técnicas e ferramentas**. 2ª Edição, Pearson Addison-Wesley