Introdução a Compilação

JavaCC - Java Compiler Compiler

Lianet Sepúlveda Torres
lisepul@icmc.usp.br

Fases de um compilador

- Analisadores léxicos
  - Quebra uma sentença em tokens e classifica estes tokens
    - Há símbolos (caracteres) ou identificadores inválidos?
- Analisadores sintáticos (Parsers)
  - Analisa uma sentença de acordo com as regras da gramática
  - Programas que recebem como entrada um arquivo fonte e diz se ele está correto sintaticamente, segundo uma ***gramática pré-definida***.

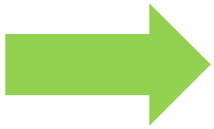
Parsers

- Há dois tipos de parsers
  - Parser Top-down
  - Parser Bottom-up
- Implementar um parser manualmente pode ser uma tarefa muito trabalhosa

Geradores de Parsers

Geradores de Parsers

Arquivo de especificação da gramática a ser aceita pelo parser



Gerador de Parser

Parser que reconhece esta gramática

JavaCC - Java Compiler Compiler

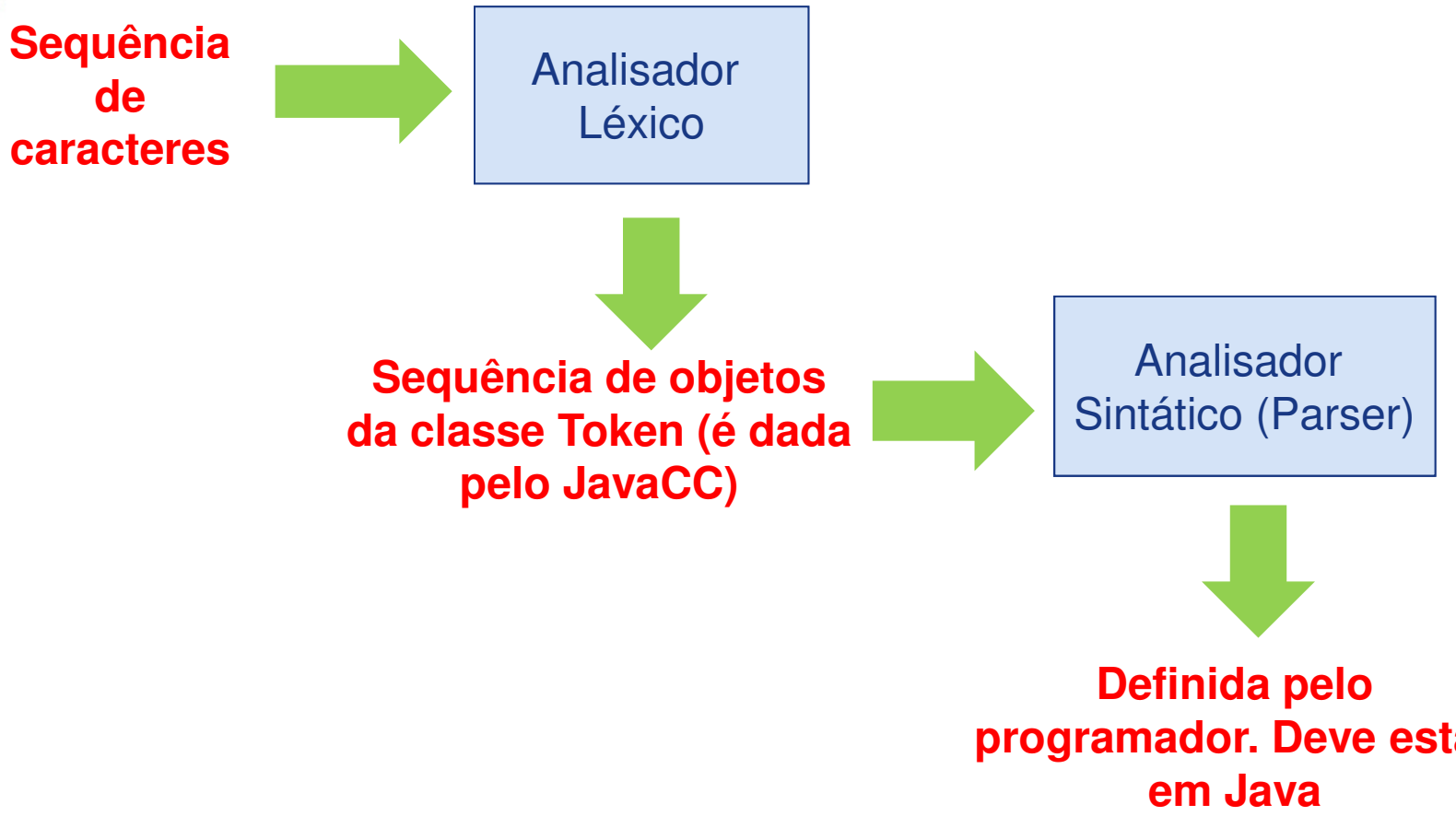
- É um gerador de analisadores léxicos e de *parsers*

- COMPILADOR / INTERPRETADOR → incorpora um analisador léxico e um *parser*
  - O **analisador semântico** e a **geração de código** são possíveis via inclusão de código JAVA

JavaCC

- 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

JavaCC


Sequência de caracteres
Analisador Léxico
Sequência de objetos da classe Token (é dada pelo JavaCC)
Analisador Sintático (Parser)
Definida pelo programador. Deve estar em Java

Saída do Analisador Sintático

- 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

JJTree e JJDoc ferramentas de JavaCC

JavaCC

- Lê uma gramática no formato **EBNF**
  - **Gramática LL(k)**

    L - ***Left-right***: sentido de leitura → da esquerda para a direita

    L – ***Leftmost derivation***: tipo de derivação considerada → derivação mais à esquerda

    (k) – ***k lookahead***: número de símbolos necessários para distinguir a produção correta

LL(k)

- **LL(1):**
  - Dada: A -> X1A1 | X2A2|...| XnAn → First(Xis) são disjuntos 2 a dois.
  - Para toda produção A-> α | β:
    - Se β =>* λ → α não deriva cadeias começando com um terminal no Follow(A)
      - First(α) ≠ Follow(A)
    - Se α =>* λ → β não deriva cadeias começando com um terminal no Follow(A)
      - First(β) ≠ Follow(A)

Algumas restrições de LL(k)

- A gramática LL(k) 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.
```

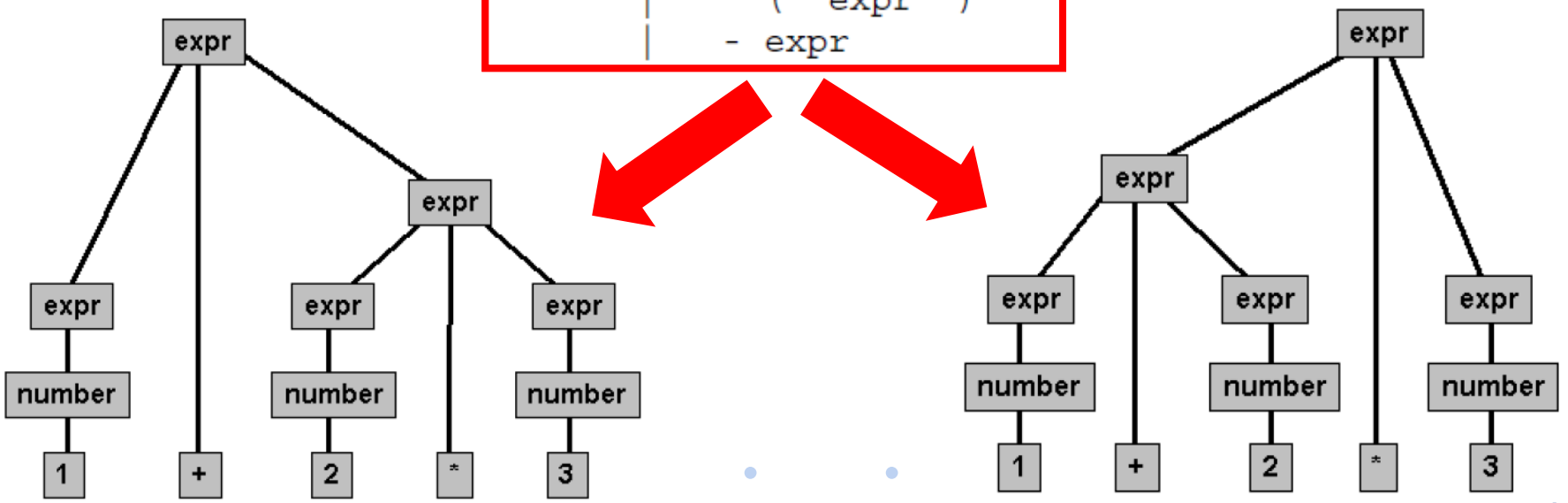
Exemplos retirados de: Using JAVACC: http://www.cs.lmu.edu/~ray/notes/javacc/

Algumas restrições de LL(k)

- A gramática LL(k) não pode ser ambígua:

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

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



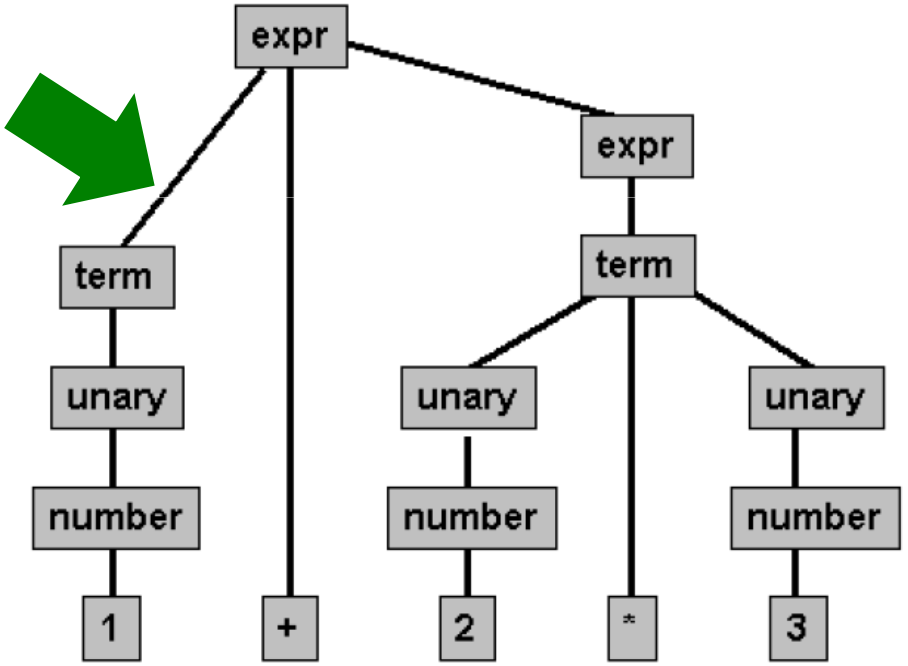
Exemplos retirados de: Building your own languages with JAVACC: http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html

Algumas restrições de LL(k)

- A gramática LL(k) 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-cooltools.html

LL(1)

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

- Exercício de casa
  - Verificar se Pascal Simplificado com as extensões é LL(1)

Instalação do JavaCC

- Instale o JAVA (jdk 6.0, por exemplo)
- Faça o download do JAVACC e descompacte: http://javacc.java.net/
- Adicione o diretório /bin do JAVACC na variável de ambiente do sistema PATH
- **JavaCC Plug-in de Eclipse** (Help/Install New Software → http://eclipse-javacc.sourceforge.net/)

Primeiro exemplo - Calculadora

- Realizar operações de soma, subtração e multiplicação com número inteiros

```
D:\Documentos\usp\PAE>java Calculator
1+2
3
-2+1
-1
```

- Exemplo de funcionamento:
  - Operação = 1+2 → Resultado = 3
  - Operação = -2+1 → Resultado = -1
- Gramática:

Start -> Expression
Expression → Term (PLUS Term | MINUS Term)*
Term → Primary (TIMES Primary)*
Primary -> <NUMBER>

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

Primeiro exemplo – Calculadora

- Definição da classe do Parser
  - Bloco para inserir declarações de java dentro da classe (podem ser definidos métodos e variáveis auxiliares)

```
PARSER_BEGIN(Calculator)
import java.io.PrintStream ;
class Calculator {
    static public void main( String[] args )
    {
        Calculator parser = new Calculator( System.in ) ;
        while (true){
            try{
                parser.Start( System.out ) ;
            }catch (ParseException ex){
                System.out.println(ex.getMessage());
                System.exit(-1);
            }catch (TokenMgrError ex){
                System.out.println(ex.getMessage());
                System.exit(-1);
            }
        }
    }
    int previousValue = 0 ;
}
PARSER_END(Calculator)
```

Classe Principal (Parser que será gerado Calculator.java)

Entrada de tokens

Método gerado pelo símbolo inicial da gramática

Classes de exceções geradas pelo JAVACC

Primeiro exemplo – Calculadora

- Definição dos Tokens (Componente Léxico)
  - Serve para especificar os tipos de tokens e as expressões regulares associadas
  - Nesta seção são descritas as palavras reservadas

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

Caracteres desconsiderados durante a análise

Tokens são definidos por ERs

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

Primeiro exemplo – Calculadora

- Definição das Produções (Componente Sintático)
  - Ações semânticas associadas

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

Cada não terminal da gramática é um método do parser

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

Código Java inserido

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

Primeiro exemplo – 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");
          }
        }
}
```

Cada não terminal da gramática é um método do parser

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

Código Java inserido

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

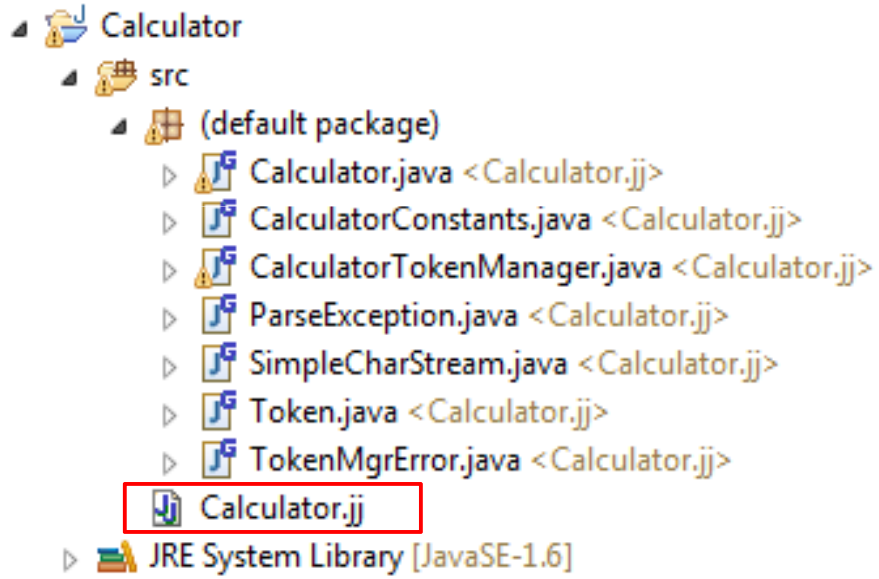
Primeiro exemplo – Calculadora

- Opções do Parser (Não obrigatório)
  - STATIC – diz se os métodos do parser serão estáticos (default: true)
  - LOOKAHEAD – informa o nível de profundidade do lookahead (default: 1)
  - DEBUG_PARSER – instrui o parser a registrar todas as informações durante o parsing do arquivo (default: false)
  - DEBUG_LOOKAHEAD – instrui o parser a registrar também as tentativas de lookahead (default: false)

```
options
{
  JDK_VERSION = "1.5";
  static = true;
}
```

JavaCC - Gerando o Parser (Calculator.jj)

- javacc Calculator.jj → Gera 7 arquivos:
  - Calculator.java -> parser
  - CalculatorConstants.java
  - CalculatorTokenManage.java
  - ParseException.java
  - SimpleCharStream.java
  - Token.java
  - TokenMgrError.java



- javac Calculator.java → compilar
- java Calculator → Executar o programa

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

JavaCC- Gerando o Parser

- Calculator.java → parser
- CalculatorTokenManage.java → analisador léxico
- TokenMgrError.java→ para detectar erros léxicos
- ParseException.java → para detectar erros sintáticos
  - Subclasse de Throwable (Exception)
- SimpleCharStream.java → responsável por adaptar a entrada para o analisador léxico
- Token.java → classe que representa os tokens
  - Image → string que representa o token
  - Kind → número que representa o tipo do token

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

Primeiro exemplo – Calculadora (Calculator.jj)

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

```
23*80
1840
-55+33
-22
-55+-10
-65
100*-1
-100
```

Primeiro exemplo – Calculadora (Calculator.jj)

- Mensagens para erros léxicos podem ser editadas no arquivo **TokenMgrError.java**

- Exemplo de erro léxico → uso do operador "/" não definido
  - Operação: 33/1 → Erro léxico!

```
33/1
Lexical error at line 5, column 3.  Encountered: "/" (47), after : ""
```

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

Primeiro exemplo – Calculadora (Calculator.jj)

- 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!

```
+33
Encountered " "+" "+ "" at line 1, column 1.
Was expecting one of:
    <EOF>
    "-" ...
    "(" ...
    <NUMBER> ...
```

Primeiro exemplo – Calculadora (Calculator.jj)

- 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

```
777766669999+33

Erro Semantico na linha 1, coluna 1: overflow
```

Primeiro exemplo – Calculadora (Calculator.jj)

- Exercício:
  - Incluir operações de negação e operações parentesadas na calculadora, com as prioridades:
    - Parênteses e negação tem a mesma prioridade de número

Primeiro exemplo – Calculadora (Calculator.jj)

- Exercício:
  - Incluir operações de negação e operações parentesadas na calculadora, com as prioridades:
    - Parênteses e negação tem a mesma prioridade de número

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

Primeiro exemplo – Calculadora (Calculator.jj)

- Exercício:

```
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 : ")" > }
```

JavaCC - 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

JavaCC - Regras de Desambiguação

```
void basic_expr() :
{}
{
  <ID> "(" expr() ")"
|
  "(" expr() ")"
|
  "new" <ID>
|
  <ID> "." <ID>
}
```

- Lookahead = 1 (default)
- Aparece o *warning* porque o JavaCC reconheceu que a gramática não é LL(1)

→ Se não mostra *warning* então a gramática é LL(1)

- Neste exemplo a gramática é ambígua

```
Warning: Choice conflict involving two expansions at
         line 25, column 3 and line 31, column 3 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
```

JavaCC - Regras de Desambiguação

```
void basic_expr() :
{}
{
  <ID> "(" expr() ")"
|
  "(" expr() ")"
|
  "new" <ID>
|
  <ID> "." <ID>
}
```

- Aparece o *warning* porque o JavaCC
- ática não é
- *rning* então
- a é

Quando aparece o *warning* → Duas soluções

```
Warning: Choice conflict involving two expansions at
         line 25, column 3 and line 31, column 3 respectively.
         A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier expansion.
```

JavaCC - Regras de Desambiguação

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

```
void basic_expr() :
{}
{
  <ID> "(" expr() ")"
|
  "(" expr() ")"
|
  "new" <ID>
|
  <ID> "." <ID>
}
```

```
void basic_expr() :
{}
{
  <ID> ( "(" expr() ")" | "." <ID> )
|
  "(" expr() ")"
|
  "new" <ID>
}
```

JavaCC - Regras de Desambiguação

- 2ª solução: Indicar ao parser quando a gramática não é LL(1)→ uso do Lookahead >1

```
void basic_expr() :
{}
{
  LOOKAHEAD(2)
  <ID> "(" expr() ")"
|
  "(" expr() ")"
|
  "new" <ID>
|
  <ID> "." <ID>
}
```

Local lookahead : Trata como LL(K) determinadas produções da gramática

Global Lookahead: Trata toda a gramática como LL(K)

Projeto → Parte 1

- **Analisador Léxico:**
  - Criar um analisador léxico que tabula a saída código/token para Pascal Simplificado com as extensões do grupo

Projeto → Parte 1

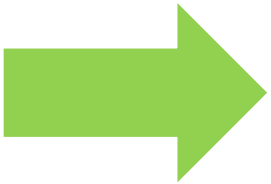
```
PARSER_BEGIN (AnaLex)
        public class AnaLex{
                public static void main(String[] args){
                        AnaLex parser = new AnaLex(System.in);
                        while(true){
                                try {
                                        parser.parser();
                                } 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)
```

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.
```

Projeto → Parte 1

Saída do Analisador Léxico



```
@(1,1)   PROGRAM      - program
@(1,9)   ID           - teste1
@(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          - .
```

Erros devem ser adaptados

Para analisador léxico → TokenMgrError.java

Referências

- **JavaCC [tm]: Documentation Index:** http://javacc.java.net/doc/docindex.html
- **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://ltiwww.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**
- Aho, A. V., Lam, M. S., Sethi, R. e Ullman, J. D. (2008): **Compiladores: Princípios, técnicas e ferramentas**. 2ªEdição, Pearson Addison-Wesley
- **Create your own Programming Languege:** http://www.codeproject.com/Articles/50377/Create-Your-Own-Programming-Language
- **JavaCC Eclipse Plug-in:** http://eclipse-javacc.sourceforge.net/