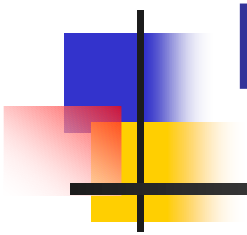


Breve Tutorial de JavaCC

O que é?

Instalação:

http://javacc.dev.java.net

Exemplos usados de:

http://w3.msi.vxu.se/users/jonasl/javacc

http://www.cs.nmsu.edu/~rth/cs/cs471/InterpretersJavaCC.html

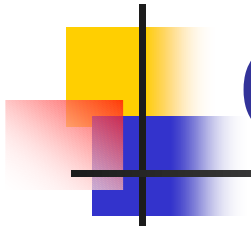
http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf

DOCS:

https://javacc.dev.java.net/doc/docindex.html

https://javacc.dev.java.net/doc/tokenmanager.html

Material de Apoio da Web



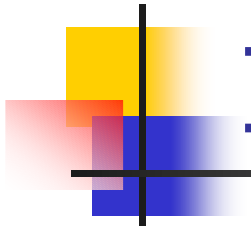
O compiler compiler JavaCC

- Gerador de parser/scanner descendente recursivo para a linguagem Java
- Versão atual: JavaCC 5.0

- JavaCC é uma ferramenta que lê a especificação da gramática em EBNF (a descrição dos tokens está no mesmo arquivo da gramática e é dada em ER) e converte ela para um programa em Java que pode reconhecer programas para a dada gramática.

- The input to the lexical analyser is a sequence of character — represented by a Java InputStream object or a Java Reader object.
- The output of the lexical analysers is fixed by JavaCC: it is a sequence of Token objects. The input to the parser is again fixed, it is a sequence of Token objects.
- The output of the parser is, however, not prescribed by JavaCC at all; it is whatever the programmer wants it to be, as long as it can be expressed in Java.

- Pode gerar uma representação intermediária na forma de uma árvore sintática abstrata via ferramenta JJTree, que acompanha o JavaCC. No nosso projeto a saída será somente a decisão do programa estar correto léxica e sintaticamente.



Instalação

- Instale Java na sua máquina (p.e. JDK 6)

- Faça download do arquivo ZIP ou GZIP de:

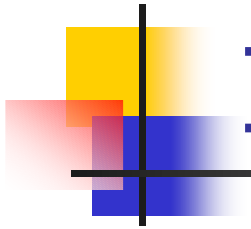
https://javacc.dev.java.net/servlets/ProjectDocumentList

e descompacte

- Add the bin directory within the JavaCC installation to your path

- The javacc, jjtree, and jjdoc invocation scripts/executables reside in this directory.

Primeiro exemplo: Soma de Inteiros



- Somar listas de números como:

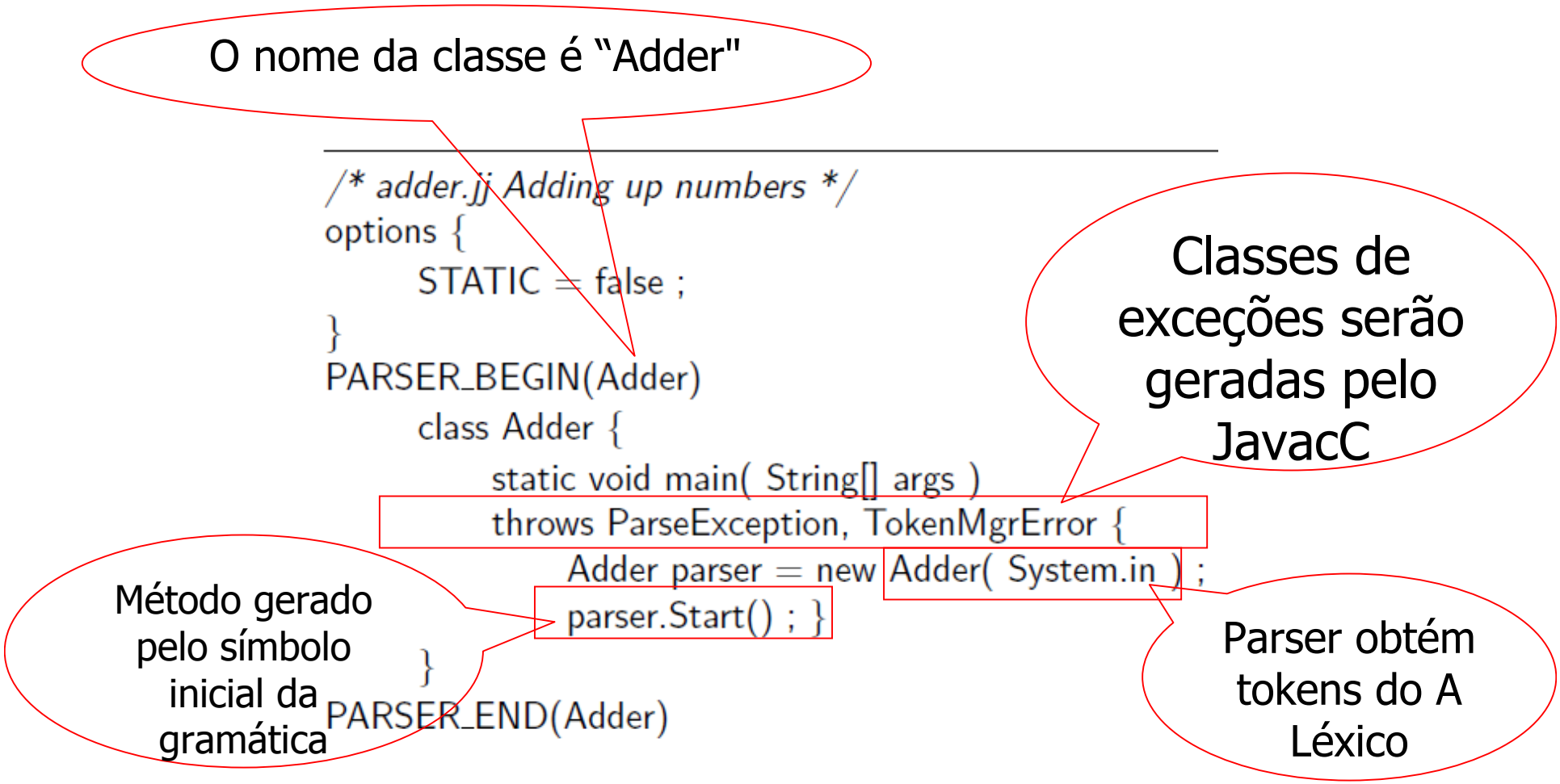
99 + 42 + 0 + 15

- Permitiremos espaços e quebras de linhas
- O arquivo de especificação para o JavaCC se chamará **adder.jj** e terá especificação da parte léxica e da gramática

- adder.jj tem 3 partes:

1. Declaração da classe e de opções

A primeira parte do arquivo contém um template da classe onde se decide as propriedades (nome, visibilidade, etc.) da classe do parser Java que será gerado. JavaCC irá aumentar esta classe no processo de geração.



First note that main might throw either of the two generated subclasses of Throwable. This is not very good style, as we really ought to catch these exceptions, however, it helps keep this first example short and uncluttered.

The first statement of the body creates a new parser object. The constructor used is automatically generated and takes an InputStream. There is also a constructor that takes a Reader. The constructor in turn constructs an instance of the generated SimpleCharacterStream class, and a lexical analyser object of class AdderTokenManager. Thus, the effect is that the parser will get its tokens from a lexical analyser that reads characters from the System.in object via a SimpleCharacterStream object.

The second statement calls a generated method called Start. For each BNF production in the specification, JavaCC generates a corresponding method in the parser class. This method is responsible for attempting to find in its input stream a match for the description of the input. In the example, calling Start will cause the parser to attempt to find a sequence of tokens in the input that matches the description

<NUMBER> (<PLUS> <NUMBER>)* <EOF>

Outro template de classe

The parser code includes a driver specified between PARSER_BEGIN and PARSER_END.

```
public class CalcParser {
  public static void main(String[] args) {
        CalcParser parser;
        try {
              parser = new CalcParser(new FileInputStream(args[0]));
        }
        catch (FileNotFoundException e) {
              System.out.println("File not found: " + args[0]);
              return;
        }
        try {
              parser.Program();
        }
        catch (ParseException e) {
              System.out.println(e);
              return;
        }
        catch (TokenMgrError e) {
              System.out.println(e);
              return;
        }
        System.out.println("Successful parse");
  }
}
```

Método gerado pelo símbolo inicial da gramática

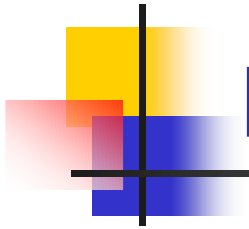
2. Especificação do A Léxico

```
SKIP : { " " }
SKIP : { "\n" | "\r" | "\r\n" }
TOKEN : { < PLUS : "+" > }
TOKEN : { < NUMBER : (["0"-"9"])+ > }
```

Nomes dos Tokens e suas ER.

JavaCC trata do EOF automaticamente

The first line says that space characters constitute tokens, but are to be skipped, that is, they are not to be passed on to the parser. The second line says the same thing about line breaks. Different operating systems represent line breaks with different character sequences; in Unix and Linux, a newline character ("\n") is used, in DOS and Windows a carriage return ("\r") followed by a newline is used, and in older Macintoshes a carriage return alone is used. We tell JavaCC about all these possibilities, separating them with a vertical bar. The third line tells JavaCC that a plus sign alone is a token and gives a symbolic name to this kind of token: PLUS. Finally the fourth line tells JavaCC about the syntax to be used for numbers and gives a symbolic name, NUMBER, to this kind of token.



Exemplos de arquivos de entrada

"123 + 456\n"

A Léxico encontra 7 tokens: um número, um espaço, um plus, outro espaço, outro número, um newline e um EOF.

Os tokens marcados com skip não são passados ao parser; ele só vê:

NUMBER, PLUS, NUMBER, EOF

"123 - 456\n"

Quando o A Léxico vê – ele gera uma exceção

Outros exemplos de A Léxico

```
/* LITERALS */
TOKEN :
{
  < INTEGER_LITERAL : "0" | ["1"-"9"] (["0"-"9"])* >
}

/* OPERATORS */
TOKEN :
{
      < PLUS: "+" >
  |   < MINUS: "-" >
  |   < MULT: "*" >
  |   < DIV: "/" >
}
```

```
TOKEN : /* numbers */
{
  < NUMBER: (<DIGIT>)+ >
|
  < #DIGIT: [ "0"-"9" ] >
}
```

ER locais ganham #

TOKEN : { < NUMBER : <DIGITS> | <DIGITS> "." <DIGITS> | <DIGITS> "." | "." <DIGITS> > }
TOKEN : { < #DIGITS : (["0"-"9"])+ > }

---

3. Especificação do parser em EBNF

```
void Start() :
{}
{
      <NUMBER>
      (
          <PLUS>
          <NUMBER>
      )*
}
```

This BNF production specifies the legitimate sequences of token kinds in error free input. The production says that these sequences begin with a NUMBER token, end with an EOF token and in-between consist of zero or more subsequences each consisting of a PLUS token followed by a NUMBER token.

As is stands, the parser will only detect whether or not the input sequence is error free, it doesn't actually add up the numbers, yet.

Gerando o parser e A Léxico

Having constructed the `adder.jj` file, we invoke JavaCC on it. Exactly how to do this depends a bit on the operating system. Below is how to do it on Windows NT, 2000, and XP. First using the "command prompt" program (CMD.EXE) we run JavaCC:

```
D:\home\JavaCC-Book\adder>javacc adder.jj
Java Compiler Compiler Version 2.1 (Parser Generator)
Copyright (c) 1996-2001 Sun Microsystems, Inc.
Copyright (c) 1997-2001 WebGain, Inc.
(type "javacc" with no arguments for help)
Reading from file adder.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.
```

This generates seven Java classes, each in its own file:

- TokenMgrError is a simple error class; it is used for errors detected by the lexical analyser and is a subclass of Throwable.
- ParseException is another error class; it is used for errors detected by the parser and is a subclass of Exception and hence of Throwable.
- Token is a class representing tokens. Each Token object has an integer field kind that represents the kind of the token (PLUS, NUMBER, or EOF) and a String field image, which represents the sequence of characters from the input file that the token represents.

- SimpleCharStream is an adapter class that delivers characters to the lexical analyser.
- AdderConstants is an interface that defines a number of classes used in both the lexical analyser and the parser.
- AdderTokenManager is the lexical analyser.
- Adder is the parser.

We can now compile these classes with a Java compiler:

```
D:\home\JavaCC-Book\adder>javac *.java
```

Exemplo de parser com ações

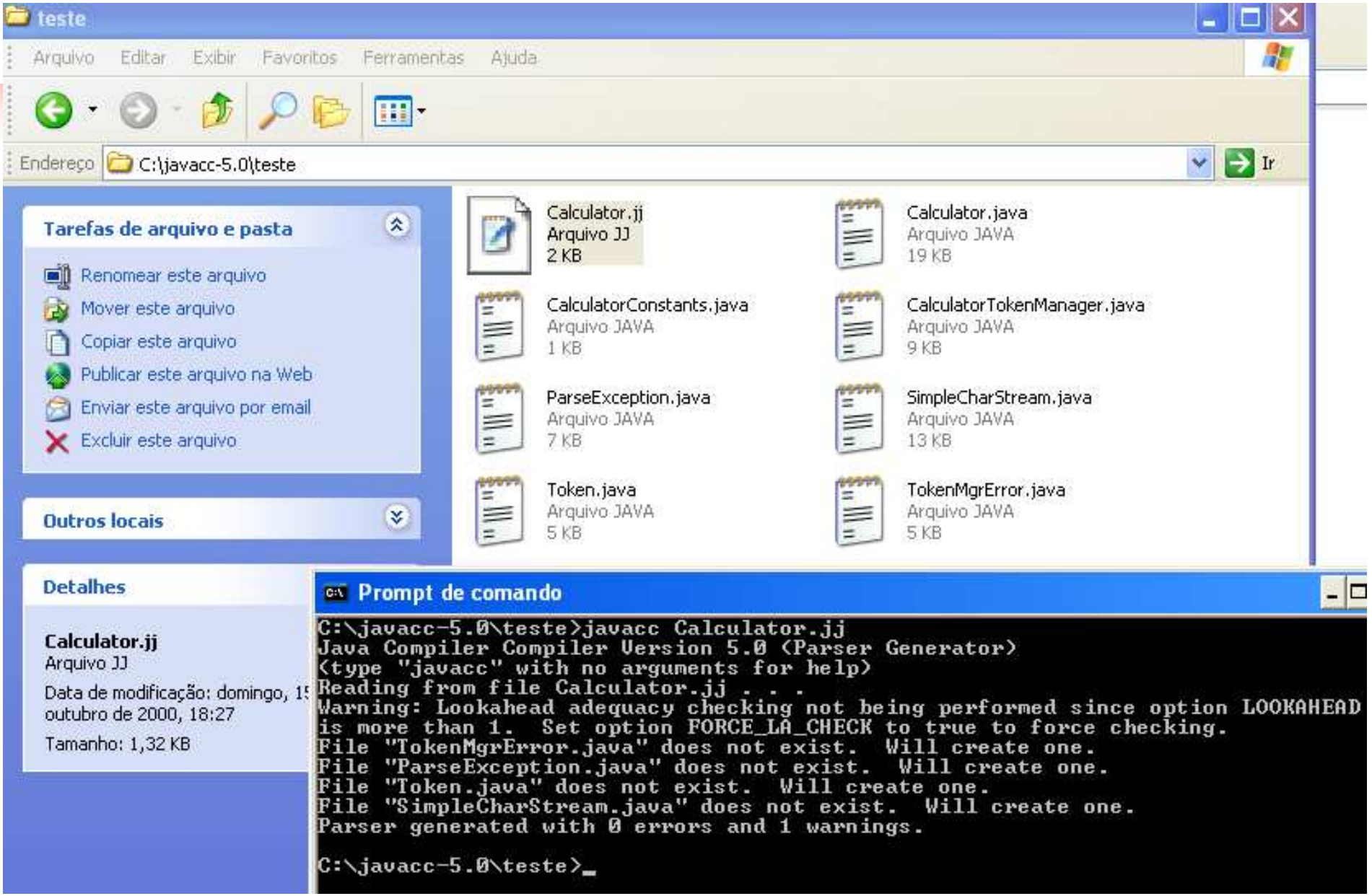
```
options
{
   LOOKAHEAD=2;
}
PARSER_BEGIN(Calculator)
public class Calculator
{
   public static void main(String args[]) throws
    ParseException
   {
      Calculator parser = new Calculator(System.in);
      while (true)
      {
         parser.parseOneLine();
      }
   }
}
PARSER_END(Calculator)
```

```
SKIP :
{
   " "
|  "\r"
|  "\t"
}
TOKEN:
{
   < NUMBER: (<DIGIT>)+ ( "."
(<DIGIT>)+ )? >
|  < DIGIT: ["0"-"9"] >
|  < EOL: "\n" >
}
```

```
void parseOneLine():
{
    double a;
}
{
    a=expr() <EOL>      { System.out.println(a); }
  | <EOL>
  | <EOF>               { System.exit(-1); }
}
double expr():
{
    double a;
    double b;
}
{
    a=term()
    (
        "+" b=expr()    { a += b; }
    |   "-" b=expr()    { a -= b; }
    )*
                        { return a; }
}
double term():
{
    double a;
    double b;
}
{
    a=unary()
    (
        "*" b=term()    { a *= b; }
    |   "/" b=term()    { a /= b; }
    )*
                        { return a; }
}
```

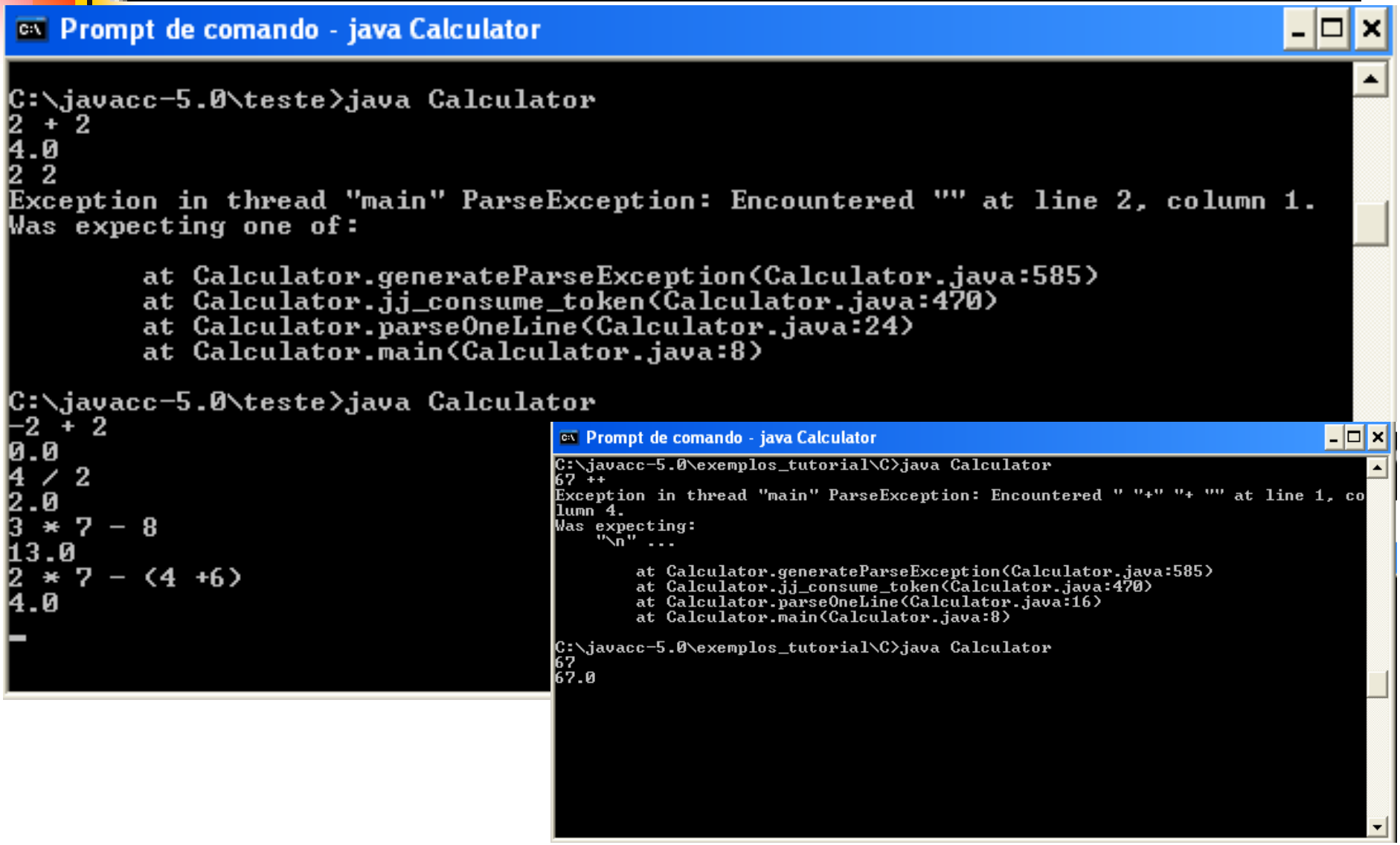
```
double unary():
{
    double a;
}
{
    "-" a=element()     { return -a; }
|   a=element()         { return a; }
}
double element():
{
    Token t;
    double a;
}
{
    t=<NUMBER>          { return Double.parseDouble(t.toString()); }
|   "(" a=expr() ")"    { return a; }
}
```

Gerando o parser: 7 arquivos

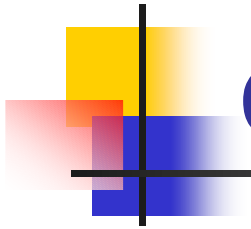


Rodar após compilar os 7

```
C:\javacc-5.0\teste>javac *.java
```

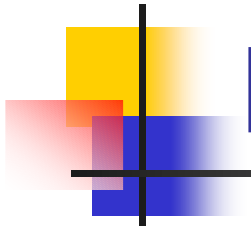


Customizando as mensagens de erros do A Léxico e Parser



- JavaCC gera dois arquivos para gerenciar os erros léxicos e sintáticos:
  - TokenMgrError.java
  - ParseException.java

- Estes podem ser editados para customização das mensagens, respeitando as partes que não podem ser alteradas.



Material de Apoio

- **Use JavaCC to build a user friendly boolean query language**

http://www.ibm.com/developerworks/data/library/techarticle/dm-0401brereton/index.html

- **Build your own languages with JavaCC**

http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html

- **JavaCC, parse trees, and the XQuery grammar, Part 1**

http://www.ibm.com/developerworks/xml/library/x-javacc1.html

- **JavaCC, parse trees, and the XQuery grammar, Part 2**

http://www.ibm.com/developerworks/xml/library/x-javacc2/