**Utilizando JavaCC - Parte 1**

**Um Analisador Léxico Simples**

**por Odilon Nelson Grisi Dantas (versão inicial em 17/08/2010)**

Ferramentas/Pré-requisitos:

Java JDK (versão 1.5 ou superior) – [http://java.sun.com](http://www.google.com/url?q=http%3A%2F%2Fjava.sun.com%2F&sa=D&sntz=1&usg=AFQjCNF5i0S7XeC7TZbUuKpVhQ2f89l5bQ)

Eclipse IDE - [http://www.eclipse.org](http://www.google.com/url?q=http%3A%2F%2Fwww.eclipse.org%2F&sa=D&sntz=1&usg=AFQjCNHjmBe6J3yE4kaaaDn62J1HeUn08A)

JavaCC - [http://javacc.dev.java.net](http://www.google.com/url?q=http%3A%2F%2Fpagesperso-orange.fr%2Feclipse_javacc%2F&sa=D&sntz=1&usg=AFQjCNFaU1OBnA95OUO-NlwnQNtzYm45kg)

E, claro, saber programar em Java, já ter utilizado o JavaCC e ter conhecimento em Construção de Compiladores ...

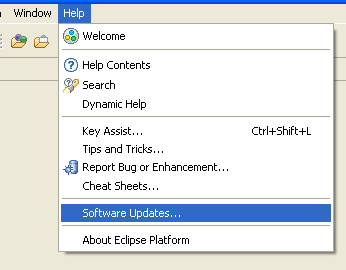
Neste tutorial, utilizaremos o plugin JavaCC para o Eclipse. Recomendo, no entanto, fazer download do JavaCC *standalone* por conter documentação e exemplos adicionais, úteis para estudo.

Neste exercício inicial, implementaremos um analisador léxico para reconhecer números inteiros em base decimal e alguns operadores.

**Mãos à obra!**

Para instalar o plugin JavaCC no eclipse, adicione a partir do *update site* [http://eclipse-javacc.sourceforge.net, conforme o passo a passo:](http://www.google.com/url?q=http%3A%2F%2Feclipse-javacc.sourceforge.net%2F&sa=D&sntz=1&usg=AFQjCNFWhhO-p11WiuHIgCjVweOtGOefsQ)

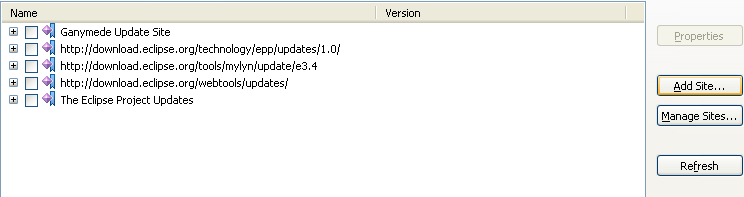
Acessar o menu *Help → Software Updates*...



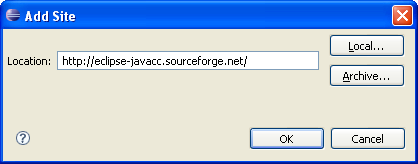
Selecionar a aba *Available Software*



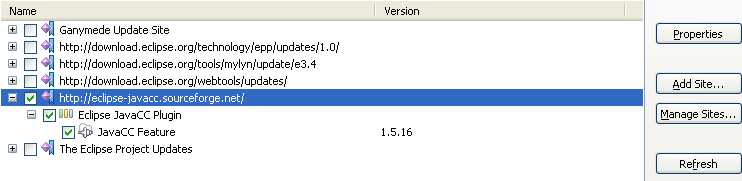
Clicar em *Add Site*



Definir o campo *Location* com o *site* [http://eclipse-javacc.sourceforge.net/](http://www.google.com/url?q=http%3A%2F%2Feclipse-javacc.sourceforge.net%2F&sa=D&sntz=1&usg=AFQjCNFWhhO-p11WiuHIgCjVweOtGOefsQ)

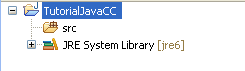


Selecionar *Eclipse JavaCC Plugin*



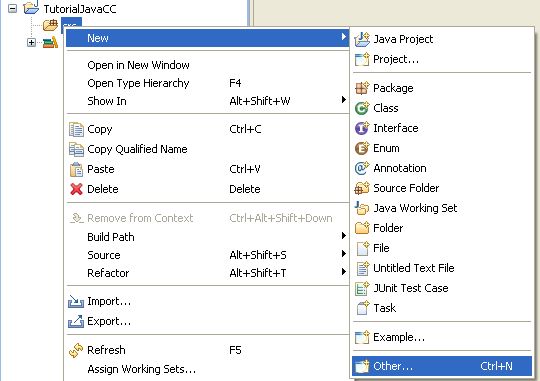
Clicar em *Install*, e aguardar o download e instalação do plugin.

Agora devemos criar um novo projeto Java no Eclipse. Nomeie-o "TutorialJavaCC".

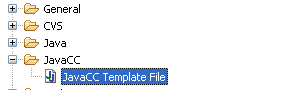


Em seguida, utilizaremos o plugin JavaCC para criar os fontes de nosso analisador.

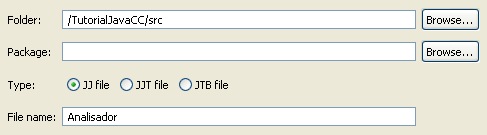
Acionar o menu *New → Other...*



No wizard, selecione a opção *JavaCC Template File*



Configurar o nome do arquivo como "Analisador.jj" e clicar em *Finish*.



O plugin gerará um arquivo "Analisador.jj" com um conteúdo inicial, como exemplo. Não utilizaremos este arquivo; apague todo o conteúdo do mesmo, substituindo pelo seguinte:

|  |
| --- |
| **options** {  **JDK\_VERSION** = "1.5";  **STATIC** = **false**;  }  **PARSER\_BEGIN**(Analisador)  **public** **class** Analisador {  }  **PARSER\_END**(Analisador)  **TOKEN** : {  < mais: "+" >  |  < menos: "-" >  |  < ponto\_virg: ";" >  }  **TOKEN** : {  < num\_int: ( ["0"-"9"] )+ >  } |

Vamos entender passo a passo cada seção do código:

options:

Nesta seção, instruímos o JavaCC a gerar código compatível com Java 1.5, e a não criar os métodos do analisador como métodos estáticos;

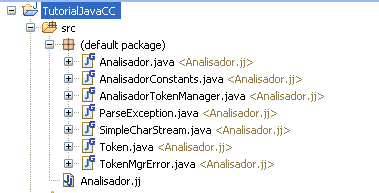
PARSER\_BEGIN / PARSER\_END:

Nesta seção, definimos a classe do nosso analisador. O nome definido nos comandos PARSER\_BEGIN e PARSER\_END deve ser igual ao nome da classe pública;

TOKEN:

Seção onde são especificadas as expressões regulares correspondentes aos tokens da linguagem.

Após um build no projeto, sua estrutura deve estar como segue (onde podemos ver as classes geradas pelo JavaCC):



**Finalizando o Analisador**

O JavaCC foi projetado para especificar as partes léxica e sintática da linguagem em conjunto. Como não temos ainda um Analisador Sintático, devemos adicionar um método ao nosso analisador para acionar a análise léxica. Devemos modificar a seção do código como segue:

|  |
| --- |
| *//código antes deste ponto não é modificado...*  **PARSER\_BEGIN**(Analisador)  **public** **class** Analisador {  **public void** processa() **throws** Exception {  **while** (**true)** {  Token t = getNextToken();  String nomeToken = tokenImage[t.kind];  System.out.println(nomeToken);  **if** (t.kind == ponto\_virg)  **break**;  }  }  }  **PARSER\_END**(Analisador)  *//código após este ponto não é modificado...* |

A modificação que fizemos foi adicionar um método *processa* que, basicamente, faz um *loop* e exibe o nome do *token* reconhecido, até que um ponto-e-vírgula seja informado.

Por questão de organização, criaremos uma classe que servirá como ponto de entrada da aplicação, invocando nosso analisador. Codifique-a como segue:

|  |
| --- |
| **public** **class** Main {  **public** **static** **void** main(String[] args) {  System.*out*.println("Digite uma expressao aritmetica");  System.*out*.print("e um ponto-e-virgula para finalizar: ");  Analisador analisador = **new** Analisador(System.*in*);  **try** {  analisador.processa();  } **catch** (Exception e) {  System.*out*.println("Ocorreu um erro.");  System.*out*.println(e.getMessage());  e.printStackTrace();  }  }  } |

Nosso *main* instancia o analisador, usando um construtor (criado pelo JavaCC) que recebe um *InputStream*, no caso, *System.in* (a entrada padrão). Chama então nosso método *processa*, capturando qualquer exceção que ocorra.

Podemos agora testar nosso analisador com qualquer combinação de números inteiros, sinais de mais ou sinais de menos. **Desde que finalizemos a entrada com um ponto-e-vírgula** (nossa condição de parada do loop), o analisador processará a contento, independente da ordem dos tokens (afinal, a fase sintática ainda não foi codificada).

Testando o analisador:

|  |
| --- |
| Digite uma expressao aritmetica  e um ponto-e-virgula para finalizar:2+2;  <num\_int>  "+"  <num\_int>  ";" |

|  |
| --- |
| Digite uma expressao aritmetica  e um ponto-e-virgula para finalizar:+-33;  "+"  "-"  <num\_int>  ";" |

O analisador léxico acusará erro caso utilizemos símbolos na entrada não definidos na nossa especificação léxica (inclusive espaços em branco, que veremos como tratar em outra ocasião).

<EOF>