# Compiladores - JFlex

Fabio Mascarenhas - 2013.1

http://www.dcc.ufrj.br/~fabiom/comp

# JFlex

- Um *gerador de analisadores léxicos* que gera analisadores escritos em Java

- A sintaxe das especificações é inspirada na sintaxe das especificações para *Lex*, um gerador de analisadores léxicos para Unix

- Baixe  em http://jflex.de/. O arquivo inclui scripts para executar ele tanto em Linux (jflex) quanto Windows (jflex.bat); use-os!

```
$ jflex scanner_spec.jflex
Reading "scanner_spec"
Constructing NFA : 36 states in NFA
Converting NFA to DFA :
............
14 states before minimization, 5 states in minimized DFA
Writing code to "Scanner.java"
```

# Especificando um Scanner

- Arquivo de especificação:

```
código Java (fica fora da classe do scanner)
%%
opções e declarações
%%
regras do scanner
```

- Código Java normalmente são *import* de pacotes que você pretende referenciar no código do scanner

- Opções controlam como é o scanner gerado

- Regras são expressões regulares e as ações que o scanner executa quando reconhece uma delas

# Opções e declarações

- **%class Foo** - Gera uma classe pro scanner com nome **Foo** (em um arquivo **Foo.java**)
- **%line** e **%column** - Ativa contagem automática de linhas e colunas, respectivamente (acessadas pelas variáveis **yyline** e **yycolumn**); útil para mensagens de erro
- **%{ ... %}** - Inclui código Java **dentro** da classe do scanner
- **%init{ ... %init}** - Inclui código Java dentro do construtor da classe do scanner
- **nome = regexp** - Define uma macro que pode ser referenciada pelas regras do scanner com **{nome}**
- **%function getToken** - Define o nome do método que executa o scanner como **getToken**
- **%type Token** (ou **%int**) - Define o tipo de retorno do método que executa o scanner como **Token**

# Expressões regulares JFlex

| Expressão | Significado |
|---|---|
| a | Caractere ‘a’ |
| “foo” | Cadeia “foo” |
| [abc] | ‘a’, ‘b’ ou ‘c’ |
| [a-d] | ‘a’, ‘b’, ‘c’ ou ‘d’ |
| [^ab] | Qualquer caractere exceto ‘a’ e ‘b’ |
| . | Qualquer caractere **exceto \n** |
| x \| y | Expressão x ou y |
| xy | Concatenação |
| x* | Fecho de Kleene |
| x+ | Fecho positivo |
| x? | Opcional |
| !x | **Negação** |
| ~x | **Tudo até x (inclusive)** |

# Regras e Ações

- Regras têm o formato

```
regexp      { código Java }
```

- O código Java é copiado para dentro do método do scanner
- Para pegar o valor do lexeme usa-se o método `yytext()`
- Lembre sempre de retornar ao final do código, ou o scanner continua rodando!
- Regra especial <<EOF>> casa com o final do arquivo

# Exemplo

```
%%

%public
%class ScannerJFlex
%function getToken
%type Token

%%

[ \n\t]    { }

[0-9]+  { return new Token(Token.NUM, yytext()); }

print    { return new Token(Token.PRINT, yytext()); }

[a-zA-Z]+     { return new Token(Token.ID, yytext()); }

[+]|[-]|;|[(]|[)]|[=] { return new Token(yytext().charAt(0), yytext()); }

<<EOF>>    { return new Token(Token.EOF, “<<EOF>>”); }

.          { throw new RuntimeException("caractere inválido "+yytext()); }
```

# Especificações heterogêneas

- O analisador léxico trabalha sem nenhuma noção da estrutura do programa, e se o próximo token que ele leu faz sentido naquela parte do programa ou não
  - Um analisador léxico para Java interpretaria `123+-/4if` como um número, seguido de +, seguido de -, seguido de /, seguido de outro número, seguido de if
- Mas o nível léxico da linguagem pode não ser uniforme
- Em HTML, por exemplo, as regras léxicas no interior de uma tag (entre os tokens < e > ou />) são diferentes das regras fora de uma tag

# Estados

- Podemos tratar uma linguagem com regras léxicas heterogêneas como várias linguagens misturadas
- Cada uma com sua especificação léxica homogênea
- Basta haver um mecanismo de separar as diferentes especificações, e chavear entre elas
- No JFlex (e todos os analisadores léxicos derivados do *Lex* original) isso é feito através de *estados*

# Estados

- Um estado é um jeito de isolar partes da especificação léxica

```
<ESTADO> {
    ... regras ...
}
```

- As regras dentro do bloco só serão válidas se o o estado atual do analisador léxico for *ESTADO*

- Uma regra fora de um bloco vale em qualquer estado

- Há sempre um estado inicial *YYINITIAL*

# Declarando e mudando estados

- Estados são declarados na seção de declarações com a diretiva **%state**

- Para mudar de um estado para outro usa-se a função `yybegin` dentro da ação de algum token, passando o estado para o qual se quer ir

  - Não existe uma função `yyend`! Para voltar a o estado inicial se usa `yybegin(YYINITIAL);`

```
%state TAG
%%
<YYINITIAL> {
  [<] { yybegin(TAG); return new Token('<'); }
  [^< ]+ { return new Token(PALAVRA, yytext()); }
}
<TAG> {
  [>] { yybegin(YYINITIAL); return new Token('>');
  ... outras regras ...
}
```

# Outros usos de estados

- Usar estados pode ser útil mesmo que a linguagem tenha uma especificação léxica homogênea
- Podemos tratar partes tradicionalmente espinhosas de muitas linguagens, como literais string e comentários, com estados próprios para isso
- Regras mais simples para o que é permitido no interior de uma string ou um comentário, sem afetar o resto da especificação
- Podemos até fazer coisas que só com expressões regulares não é possível!

# Comentários aninhados

- Um comentário Java é qualquer texto entre `/*` e `*/`
  - Ou seja, o primeiro */ que aparece dentro de um comentário acaba ele!
  - Em `/* foo /* bar */ baz */` o comentário termina no primeiro `*/`, e `baz */` vai ser tokenizado normalmente
- Mas existem linguagens que permitem aninhamento de comentários, onde o que está acima seria um único comentário
- Não podemos expressar a linguagem dos comentários aninhados com uma expressão regular (lema do bombeamento), mas podemos simular isso com estados e ações em JFlex

# Comentários aninhados

- A ideia é ter um estado só para comentários, e as outras regras do scanner ficam associadas apenas ao estado YYINITIAL
- Quando entramos no estado de comentários, inicializamos um contador de nível de aninhamento em 1
- Dentro do estado de comentários, cada /* encontrado aumenta nosso nível de aninhamento em 1
- Cada */ encontrado diminui o nível em 1, e quando o nível chega a 0 voltamos a YYINITIAL
- Qualquer outro caractere, incluindo quebras de linha, é ignorado

# Mudando o estado em outras partes

- A mudança de um estado para outro pode não ser controlada pela análise léxica, mas por outras partes do compilador

- Em Java, `List<List<Integer>> foo` é uma sequência ID < ID < ID > > ID se isso for uma declaração de uma variável, enquanto é ID < ID < ID RSHIFT ID se isso for uma expressão

    - O analisador sintático pode mudar o estado do analisador léxico a depender de qual parte do programa ele está analisando

# Outros truques com JFlex - Indentação

- A linguagem Python não tem tokens especiais para delimitar blocos no programa

- Ela usa *indentação* para sinalizar um bloco, aproveitando que é bastante comum agrupar todas os comandos de um bloco em um mesmo nível de indentação

```
def fatorial(n):
    res = 1
    while n > 1:
        res = res * n
        n = n - 1
    return res

print fatorial(5)
```

# Indentação

- A ideia é manter uma pilha de níveis de indentação, onde cada nível é o número de espaços daquele nível

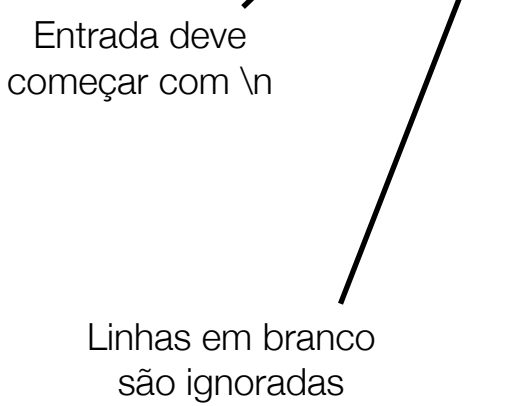
- Então associamos uma expressão regular que casa espaços no início de cada linha a uma regra que:

  - Empilha um novo nível de indentação caso o número de espaços seja maior que o topo da pilha, e gera um token BEGIN

  - Não faz nada se o número de espaços seja igual ao topo da pilha

  - **Desempilha e gera um token END enquanto o número de espaços é menor que o topo da pilha**

# Indentação




```
\n[ ]*. {
          int nivel = yytext().length() - 2;
          int atual = niveis.peek();
          yypushback(1); // volta o .
          if(nivel > atual) {
            // indenta
            niveis.push(nivel);
            return new Token(BEGIN);
          } else if(nivel < atual) {
            niveis.pop();
            // vai casar de novo para
            // gerar todos os ENDs
            yypushback(nivel + 1);
            return new Token(END);
          } else {
            // mesmo nível, não faz nada!
          }
        }

<<EOF>> {
          if(niveis.peek() > 0) {
            levels.pop();
            return new Token(END);
          } else {
            return new Token(Token.EOF, "");
          }
        }
```