Linguagens de Domínio Específico

Fabio Mascarenhas – 2017.1

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

Gramática da DSL de máquina de estados

```
machine    := events revents? commands state+
events     := "events" event+ "end"
event      := nome codigo
commands   := "commands" command+ "end"
command    := nome codigo
revents    := "resetEvents" nome+ "end"
state      := "state" nome actions? transition* "end"
actions    := "actions" "{" nome+ "}"
transition := nome "=>" nome
```

```
reservada   := 'events' | 'state' | 'end' | 'action'
nome        := !reservada [a-z][a-zA-Z]*
codigo      := [A-Z0-9][A-Z0-9][A-Z0-9][A-Z0-9]
comentario  := '--'[^\n]*
espaco      := [ \n\r\t]+ | comentario
```

Melhorias

- Não é difícil fazer o analisador ler caracteres a partir de um *Reader* qualquer ao invés de uma string
- Às vezes um único caractere não é suficiente para determinar qual o próximo token (é o caso dos comentários na nossa DSL de máquina de estados); nesse caso, precisamos examinar mais de um caractere à frente (*peek*)
- Java tem decoradores `LineNumberReader`, que fornece números de linha, e `PushbackReader`, que permite ler caracteres e depois por eles “de volta” na entrada

Comentários aninhados

- Caso um terminal apareça em uma regra, podemos chamar seu método

- Isso e útil para permitir aninhamento de comentários, por exemplo:

```
comment := '/*'(comment | .)*'*/'
```

```
void comment() {
  match('/');
  match('*');
  while(c != '*' || peek(1) != '/') {
    if(c == '/' && peek(1) == '*') {
      comment();
    } else skip();
  }
  match('*');
  match('/');
}
```

Abstração

- Ao invés de criar métodos especializados para classes de caracteres, podemos extrair e generalizar classes com *predicados*

```
public interface Predicate<T> {
  boolean test(T x);
}
```

- Dado um predicado, não é difícil fazer métodos que consomem um caractere que passa naquele predicado, ou consomem zero ou mais (um ou mais) caracteres daquele predicado

- Operações como sequenciamento e escolha também podem ser abstraídas, vamos ver isso mais adiante com a *análise por combinadores*

Análise Sintática Descendente

- O analisador sintático descendente é parecido com o analisador léxico, mas trabalhado com tokens e as regras gramaticais ao invés de caracteres e as regras léxicas
- Ainda usamos o *lookahead* para escolhas, mas o lookahead agora é um token
- O método `match` tenta consumir um token de um tipo específico, e usamos ele para os terminais
- Uma escolha usa o token de lookahead como índice de um *switch-case*, ou para testes em um if em cascata, testando se o token de lookahead prevê aquela alternativa ou não

Escolhas

- Uma escolha vira ifs ou switch/cases

```
(«alt1»|«alt2»|..|«altN»)
```

```
if ( «lookahead-predicts-alt1» ) { «match-alt1» }
else if ( «lookahead-predicts-alt2» ) { «match-alt2» }
...
else if ( «lookahead-predicts-altN» ) { «match-altN» }
else «throw-exception» // parse error (no viable alternative)
```

```
switch ( «lookahead-token» ) {
    case «token1-predicting-alt1» :
    case «token2-predicting-alt1» :
    ...
        «match-alt1»
        break;
    case «token1-predicting-alt2» :
    case «token2-predicting-alt2» :
    ...
        «match-alt2»
        break;
    ...
    case «token1-predicting-altN» :
    case «token2-predicting-altN» :
    ...
        «match-altN»
        break;
    default : «throw-exception»
}
```

Opcional e repetição

- Opcional vira um teste

```
if ( «lookahead-is-T» ) { match(T); } // no error else clause
```

- Repetição com + vira um laço do-while

```
do {
    «code-matching-alternatives»
} while ( «lookahead-predicts-an-alt-of-subrule» );
```

- Repetição com * vira um laço while

```
while ( «lookahead-predicts-an-alt-of-subrule» ) {
    «code-matching-alternatives»
}
```

Achando conjuntos de lookahead

- Formalmente, conjuntos de lookahead são calculados a partir dos conjuntos FIRST e FOLLOW das alternativas, mas existem algumas heurísticas simples que cuidam da maior parte dos casos

- A mais simples: se uma alternativa começa com um token, o conjunto de lookahead dela é aquele token

```
stat: 'if' ...    // lookahead set is {if}
    | 'while' ... // lookahead set is {while}
    | 'for' ...   // lookahead set is {for}
    ;
```

- O conjunto de lookahead de uma escolha é a união dos conjuntos de todas as suas alternativas, e o conjunto de lookahead de um termo sintático é o conjunto do lado direito de sua regra

```
body_element
    : stat      // lookahead is {if, while, for}
    | LABEL ':' // lookahead is {LABEL}
    ;
```

Lookahead para opcional e repetição

- O lookahead é mais complicado quando temos alternativas vazias, ou explicitamente ou implicitamente

- Todo opcional e repetição tem uma alternativa vazia implícita

- Nesse caso, o mais simples é ignorar o caso vazio, e tratar ele "por eliminação": seu conjunto de lookahead é tudo que não está no conjunto de lookahead do termo opcional ou repetido

- Quando isso não é possível, podemos ver o conjunto de lookahead do que segue a opção ou repetição

```
/** Match -3, 4, -2.1 or x, salary, username, and so on */
expr: '-'? (INT|FLOAT) // '-', INT, or FLOAT predicts this alternative
    | ID               // ID predicts this alternative
    ;
```

Interseção dos conjuntos

- A análise sintática descendente assume que os conjuntos de lookahead das alternativas de uma escolha são *disjuntos*

- Quando isso não acontece, podemos ter um bug na gramática, ou simplesmente uma gramática que precisa de uma técnica mais poderosa

```
expr: ID '++'    // match "x++"
    | ID '--'    // match "x--"
    ;
```

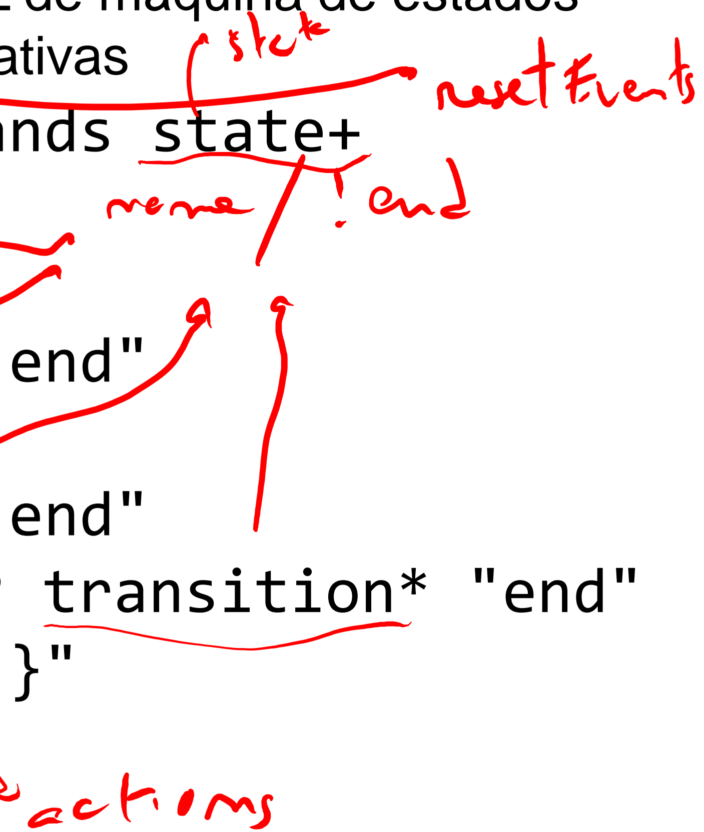
- Às vezes podemos resolver esse problema *adiando* a decisão, o que é equivalente a *fatorar* a gramática

```
expr: ID ('++'|'--') ;   // match "x++" or "x--"
```

Máquina de estados – lookahead

- Os conjuntos de lookahead para a gramática da DSL de máquina de estados são simples de calcular, já que são poucas as alternativas

```
machine    := events revents? commands state+
events     := "events" event+ "end"
event      := nome codigo
commands   := "commands" command+ "end"
command    := nome codigo
revents    := "resetEvents" nome+ "end"
state      := "state" nome actions? transition* "end"
actions    := "actions" "{" nome+ "}"
transition := nome "=>" nome
```



- O analisador sintático descendente é bem direto de escrever