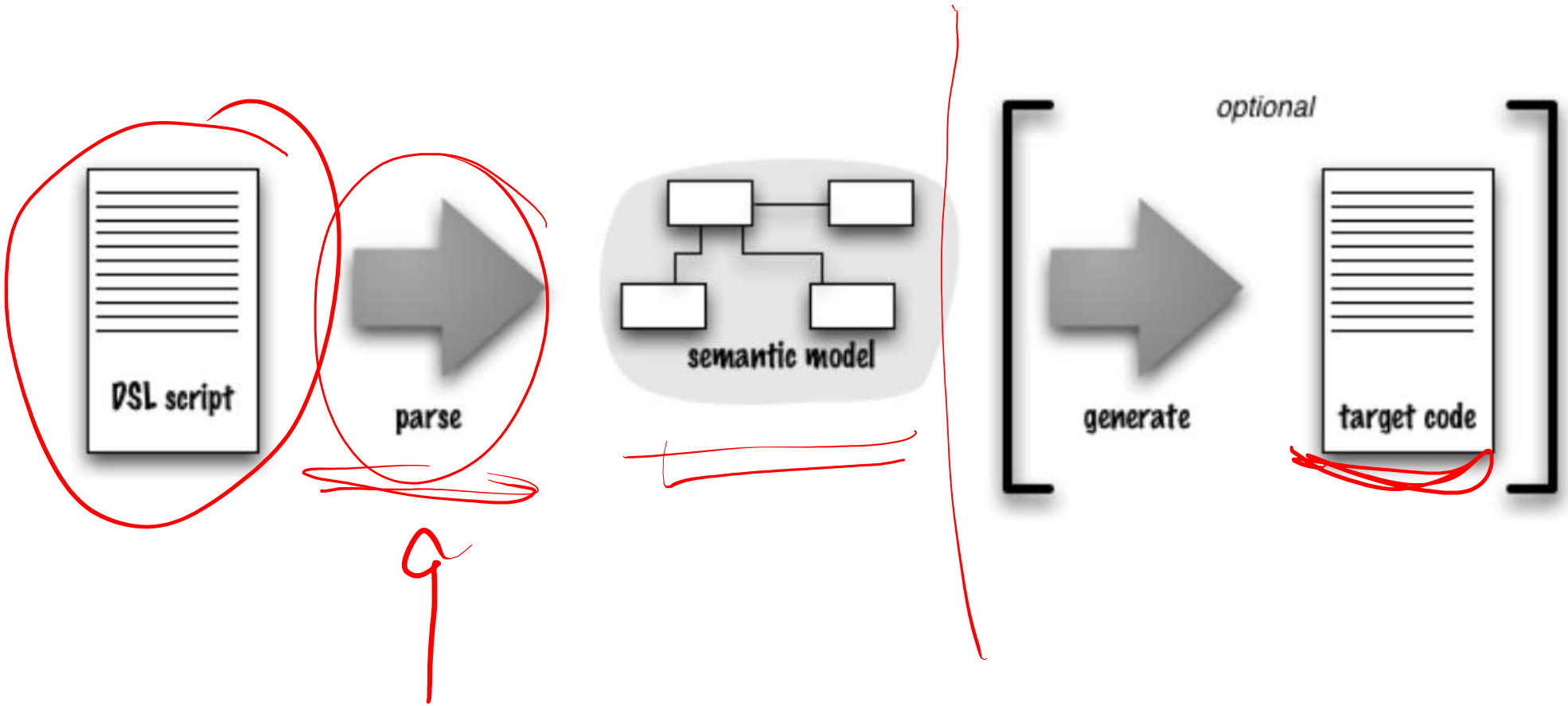
Linguagens de Domínio Específico

Fabio Mascarenhas – 2016.1

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

Processamento de uma DSL



Gramáticas

- Vamos usar uma notação parecida com a de expressões regulares para definir uma gramática

```
machine    := events resetEvents commands state+
events     := "events" event+ "end"
event      := name code
commands   := "commands" command+ "end"
command    := name code
state      := "state" name actions? transition* "end"
actions    := "action" '{' name+ '}'
transition := name '=>' name
```

- Do lado esquerdo temos termos sintáticos, e do lado direito a definição da estrutura desses termos

- Termos entre aspas e termos que não aparecem no lado esquerdo de uma regra são *terminais* ou *tokens*

Regras léxicas

- Uma gramática também precisa definir qual a estrutura dos tokens que não são simples palavras-chave ou operadores

- Podemos defini-los como parte da própria gramática, usando mais um operador tirado de expressões regulares: *classes de caracteres*

```
name := [a-zA-Z]+
code := [A-Z0-9]+
```

- Uma *classe* `[abx]` denota o conjunto { ‘a’, ‘b’, ‘x’ }

- Uma classe `[ab-fx]` denota { ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘x’ }

- Uma classe `[^ab-fx]` denota o *conjunto complemento* da classe `[ab-fx]` em relação ao conjunto de todos os caracteres

Analisador léxico descendente

- Um analisador léxico (ou *scanner*, ou *lexer,* ou *tokenizador*) agrupa os caracteres do programa em uma sequência de tokens, jogando fora espaços em branco e comentários

- Cada token é um objeto com três atributos básicos: seu *tipo*, seu *texto*, e sua *localização*

- Um analisador léxico descedente é uma forma bem simples de se codificar diretamente um analisador léxico

- A ideia é transformar a regra léxica de cada token em um método ou função para ler aquele token específico, e então ter um método *proximoToken* que, depois de pular espaços em branco, examina o próximo caractere e decide, com base nele, qual método chamar

Estrutura básica

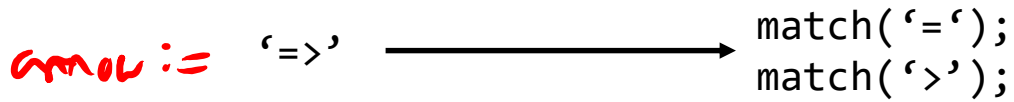
```
public abstract class Lexer {
    public static final char EOF = (char)-1; // represent end of file char
    public static final int EOF_TYPE = 1;    // represent EOF token type
    String input; // input string
    int p = 0;    // index into input of current character
    char c;       // current character
    public Lexer(String input) {
        this.input = input;
        c = input.charAt(p); // prime lookahead
    }
    /** Move one character; detect "end of file" */
    public void consume() {
        p++;
        if ( p >= input.length() ) c = EOF;
        else c = input.charAt(p);
    }

    /** Ensure x is next character on the input stream */
    public void match(char x) {
        if ( c == x) consume();
        else throw new Error("expecting "+x+"; found "+c);
    }
    public abstract Token nextToken();
    public abstract String getTokenName(int tokenType);
}
```

lookahead

Regras

- Cada caractere de uma sequência vira uma chamada para o método match



- Uma classe de caracteres vira uma chamada a um método match especializado para usar um predicado que testa se o caractere é parte daquela classe

- Uma repetição vira um laço do-while (+) ou while (*), onde usamos na condição um teste que examina o lookahead e verifica se podemos continuar a repetição

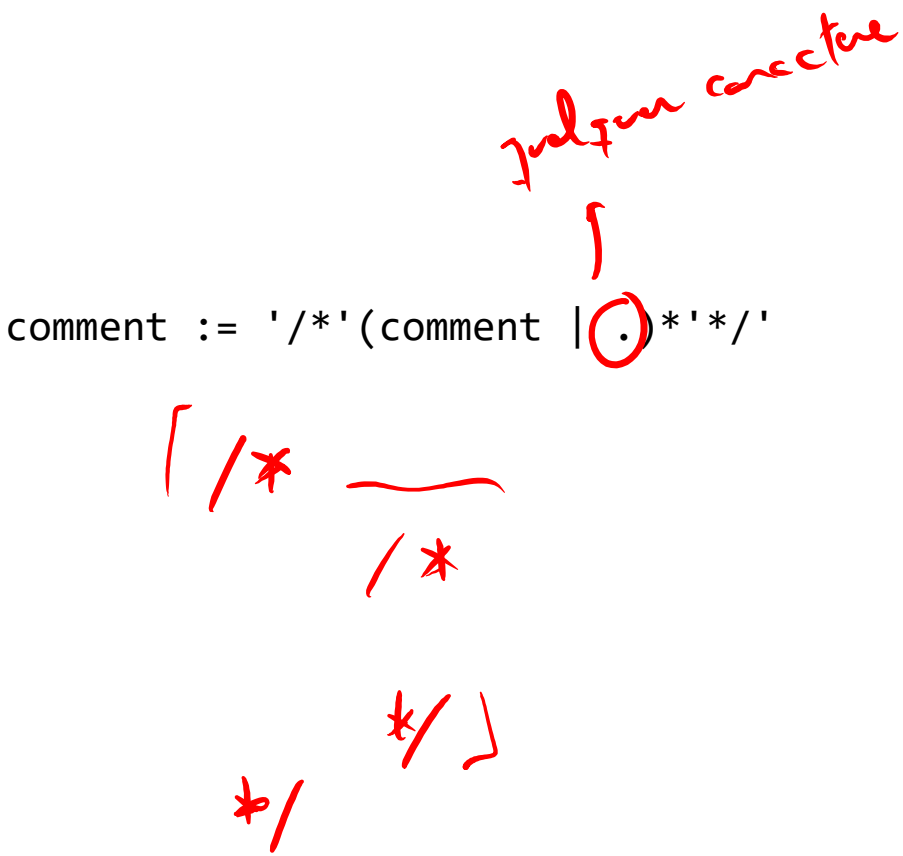
- Um opcional vira um teste condicional baseado no lookahead, e uma escolha vira um teste do lookahead para selecionar qual alternativa (parecido com o de *proximoToken*)

Melhorias

- Não é difícil fazer o analisador ler caracteres a partir de um *Reader* qualquer ao invés de uma string
- Podemos também manter internamente um contador que indica em qual linha e qual coluna da entrada estamos, para poder incluir esta informação nos tokens
- 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:




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

- 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»)

```
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»
}
```

```
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)
```

Opcional e repetição

- Opcional vira um teste

```
if ( «lookahead-is-X» ) { match(X); } // 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 commands state+
events     := "events" event+ "end"
event      := name code
commands   := "commands" command+ "end"
command    := name code
state      := "state" name actions? transition* "end"
actions    := "action" '{' name+ '}'
transition := name '=>' name
```

- O analisador sintático descendente é bem direto de escrever