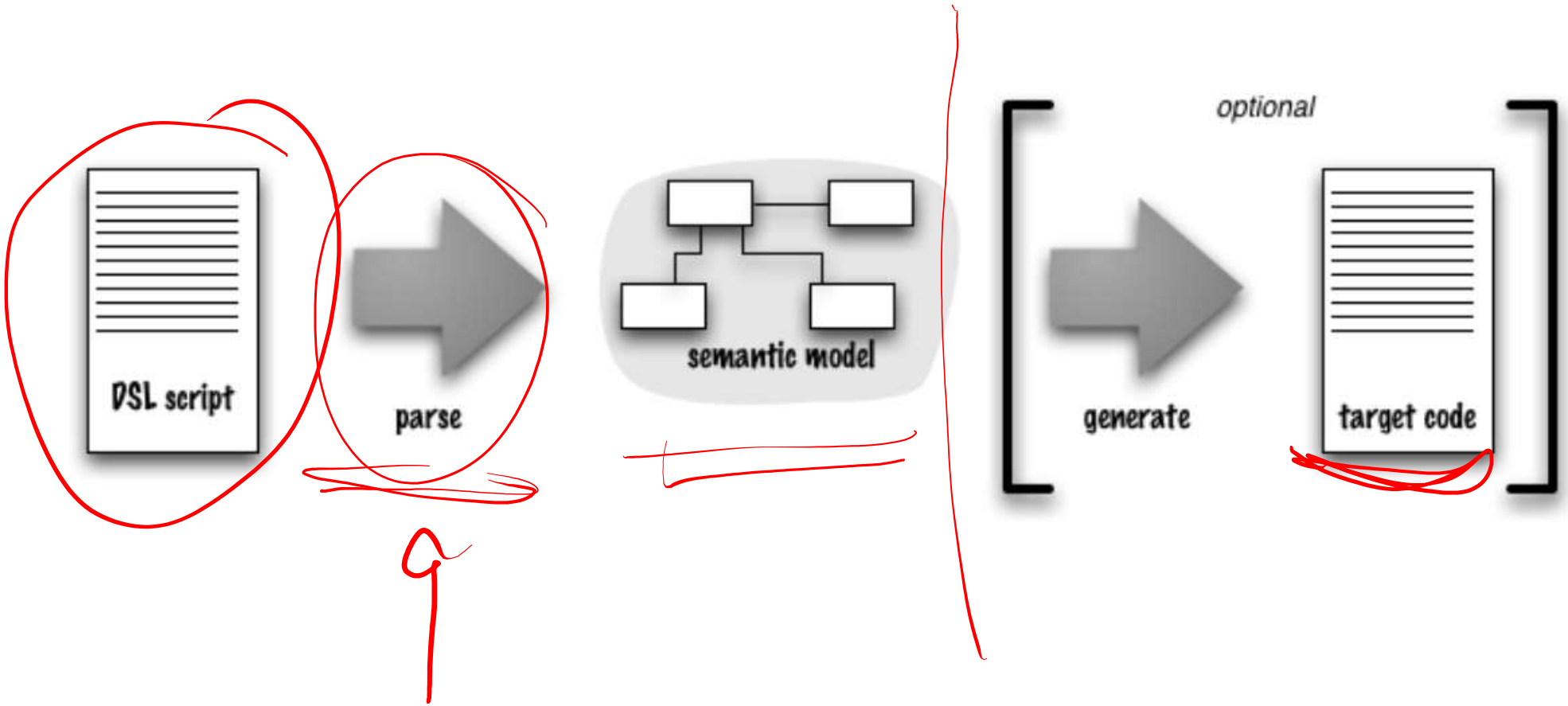
Linguagens de Domínio Específico

Fabio Mascarenhas – 2016.1

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

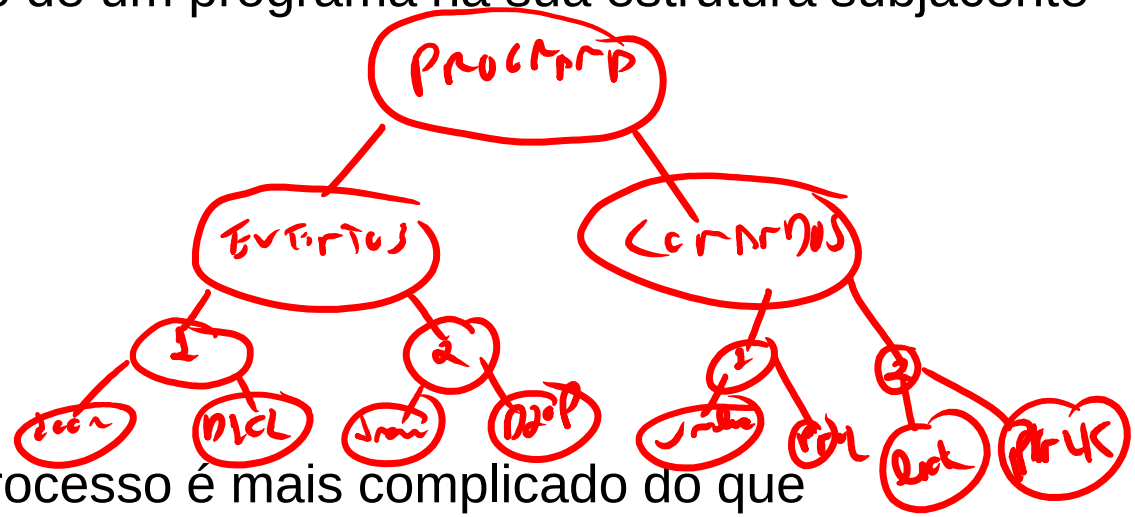
Processamento de uma DSL



Análise sintática

- A análise sintática quebra o texto de um programa na sua estrutura subjacente

```
events
  doorClosed D1CL
  drawerOpened D2OP
end
commands
  unlockPanel PNUL
  lockPanel   PNLK
end
```



- Em qualquer linguagem real o processo é mais complicado do que simplesmente quebrar em linhas e palavras e tentar interpretá-las de maneira *ad-hoc*

- O primeiro passo para criar um analisador sintático é definir uma *gramática formal* para a nossa linguagem

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*

Opcional, repetição, escolha

- Em uma gramática, o operador ? indica que o termo imediatamente anterior é opcional

- O operador * indica que o termo imediatamente anterior pode ser repetido quantas vezes quiser (até mesmo nenhuma), enquanto + exige ao menos uma repetição

- Não usamos ele na gramática de eventos, mas também temos uma operação de escolha |, que indica que podemos usar tanto a sequência de termos à esquerda do operador quanto a sequência à direita do operador

- Podemos também usar parênteses para mudar a forma como esses operadores se associam aos termos

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

Espaços em branco

- Precisamos definir também como a linguagem lida com *espaços em branco*
- Geralmente eles são ignorados, então antes de cada token implicitamente podemos ter um número arbitrário de espaços em branco que não fazem parte daquele terminal
- Outras linguagens podem ter regras mais complexas, por exemplo dando significado para quebras de linha, e/ou espaço espaços em branco no início de uma linha
- Podemos também definir qual a sintaxe dos *comentários* na linguagem, que também são considerados como espaço em branco e ignorados

Palavras reservadas

- Outra decisão de projeto é se palavras-chave na linguagem são *reservadas*, ou seja, nunca podem ser consideradas um simples identificador

- Ter palavras reservadas simplifica o analisador sintático; podemos quebrar o problema de análise sintática em dois problemas mais simples: agrupar caracteres em tokens e agrupar tokens em estruturas sintáticas

- Sem palavras reservadas precisamos de usar uma estratégia de análise mais poderosa, como técnicas baseadas em *retrocesso*

```
reserved := "events" | "state" | "end" | "action"
name     := !reserved [a-zA-Z]+
code     := !reserved [A-Z0-9]+
comment  := '--'[^\n]*
```

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, depos de pular espaços em branco, examina o próximo caractere e decide, com base nele, qual método chamar

proximoToken

- O método *proximoToken* examina o próximo caractere (chamado de *lookahead*) para decidir o que fazer

```
public Token nextToken() {
    while ( «lookahead-char»!=EOF ) { // EOF==-1 per java.io
        if ( «comment-start-sequence» ) { COMMENT(); continue; }
        ... // other skip tokens
        switch ( «lookahead-char» ) { // which token approaches?
            case «whitespace» : { consume(); continue; } // skip
            case «chars-predicting-T1» : return T1(); // match T1
            case «chars-predicting-T2» : return T2();
            ...
            case «chars-predicting-Tn» : return Tn();
            default : «error»
        }
    }
    return «EOF-token»; // return token with EOF_TYPE token type
}
```

- Palavras reservadas são tratadas como um caso especial da regra léxica para nomes, usando um conjunto de palavras reservadas

Estrutura básica

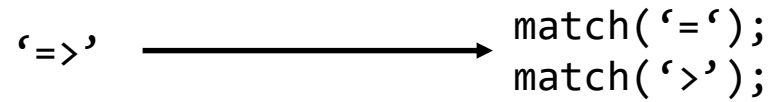
```
public abstract class Lexer {
    public static final char EOF = (char)-1; // represent end of file char
    public static final int EOF_TYPE = 1;    // represent EOF token type
    String input; // input string
    int p = 0;    // index into input of current character
    char c;       // current 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*)