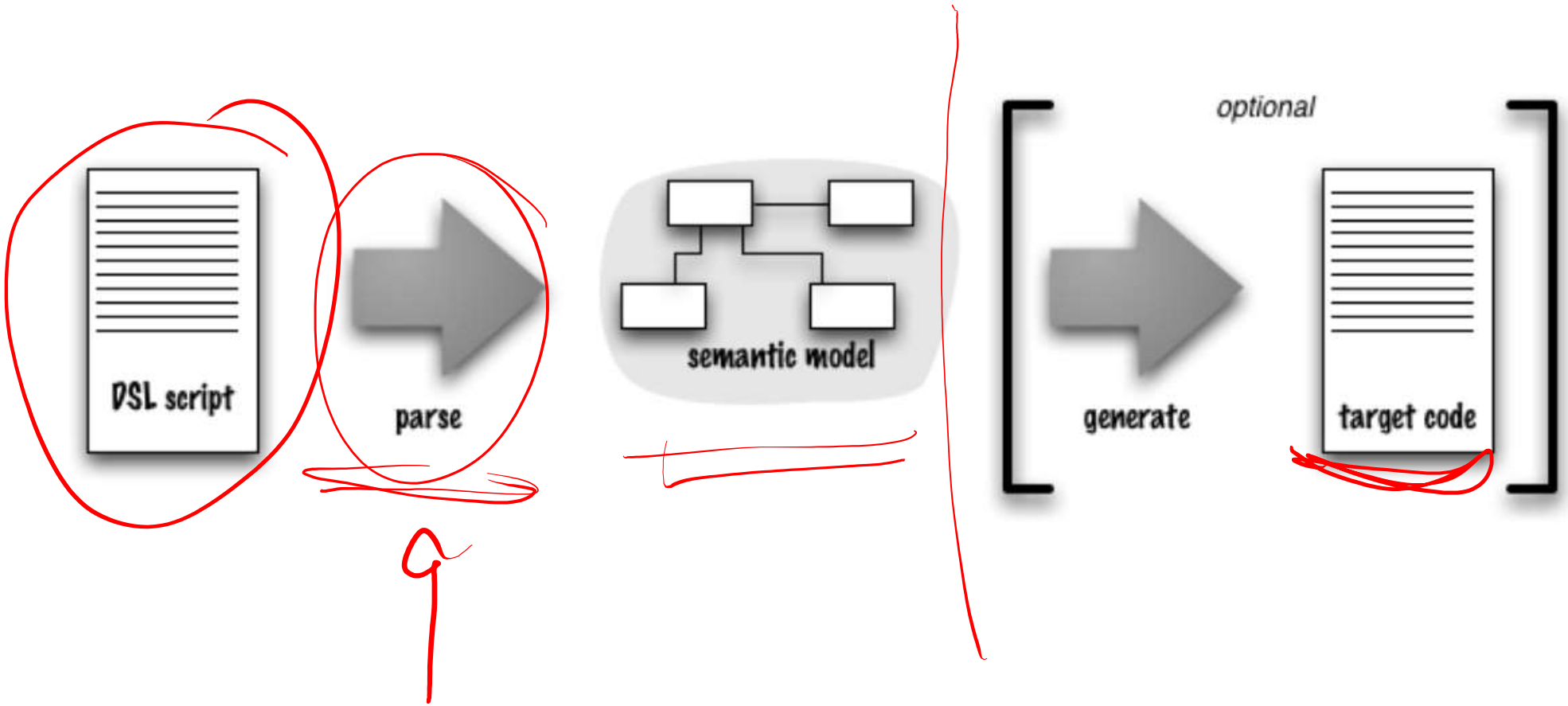
Linguagens de Domínio Específico

Fabio Mascarenhas – 2016.1

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

Processamento de uma DSL



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-I» ) { match(I); } // 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
```

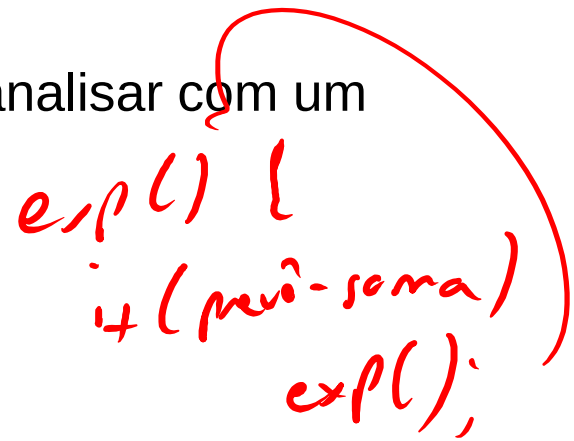
- O analisador sintático descendente é bem direto de escrever

Analisando expressões algébricas

- Nossa gramática de máquinas de estado não tem expressões algébricas, mas muitas linguagens têm

- Uma gramática de expressões ingênua é complicada de analisar com um analisador recursivo

```
exp: exp '+' exp
   | exp '-' exp
   | exp '*' exp
   | exp '/' exp
   | '(' exp ')'
   | NUM | NAME
```



- A recursão à esquerda faz ele entrar em loop, mesmo se ignorarmos o problema do lookahead

Ambiguidade

10 - 5 - 6

(1

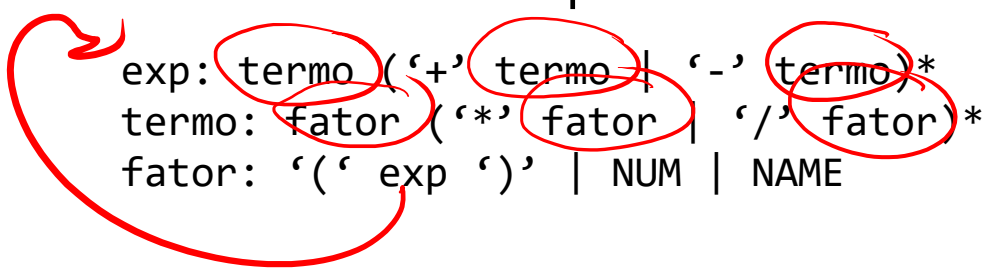
- Qual a ordem de precedência dos operadores na gramática do slide anterior? E sua associatividade?
- A gramática não define nenhum dos dois pois ela é *ambígua;* para transformar ela em uma gramática adequada para um analisador recursivo precisamos resolver essa ambiguidade
- Na linguagem anterior isso é óbvio: a multiplicação e divisão têm precedência maior que a soma e subtração, que têm a mesma precedência
- A subtração associa à esquerda, logo a soma também tem que associar, por causa da mesma precedência
- O mesmo para divisão e multiplicação

Gramática de expressões LL(1)

- Cada nível de precedência na nossa gramática ganha um não-terminal, com as expressões atômicas sendo o nível mais alto; o nível mais baixo fica com o não-terminal `exp` original

- Dentro de cada nível de expressão binária, os termos das expressões viram uma repetição, com uma escolha distinguindo entre operações de mesma precedência

- Cada nível de precedência referencia o próximo



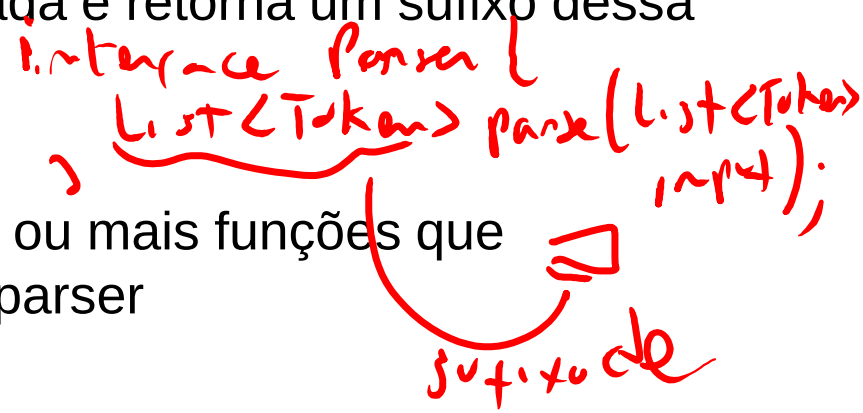
Combinadores de parsing

- Combinadores de parsing são uma técnica para expressar parsers recursivos em uma linguagem com funções anônimas ou objetos

- A ideia é construir parsers mais complexos a partir da composição de parsers mais simples, mas usando *combinadores* ao invés da composição sintática de um analisador recursivo tradicional

- Um parser é uma função que recebe uma entrada e retorna um sufixo dessa entrada caso reconheça uma parte dela

- Um combinador é uma função que recebe uma ou mais funções que descrevem parsers e as combina em um novo parser



Lista de resultados

- Nem sempre um parser é bem sucedido
- Uma determinada entrada também pode ter mais de uma análise possível
- Para representar essas duas possibilidades definimos que um parser retorna uma *lista de resultados* ao invés de um resultado só List<List<Token>>
- Cada elemento dessa lista é um sufixo da entrada original
- Se a lista for vazia, o parser falhou

Um combinador simples

- No domínio da análise sintática, os parser mais simples são aqueles que reconhecem um único token

- O combinador `token` retorna um desses parsers, dado o tipo do token desejado:

```
public class TokenParser implements Parser {
  public int tipo;
  public TokenParser(int _tipo) {
    tipo = _tipo;
  }
  public List<List<Token>> parse(List<Token> input) {
    Token tok = input.get(0);
    ArrayList<List<Token>> result = new ArrayList<List<Token>>();
    if(tok.tipo == tipo) {
      result.add(input.subList(1, list.size());
    }
    return result;
  }
}
```

Seq

- O combinador `seq` faz a sequência de dois parsers:

```
public class Seq implements Parser {
  public Parser p1;
  public Parser p2;
  public Seq(Parser _p1, Parser _p2) {
    p1 = _p1; p2 = _p2;
  }
  public List<List<Token>> parse(List<Token> input) {
    List<List<Token>> res1 = p1.parse(input);
    ArrayList<List<Token>> result = new ArrayList<List<Token>>();
    for(List<Token> suf: res1) {
      result.addAll(p2.parse(suf));
    }
    return result;
  }
}
```

Quiz

- O que acontece se o primeiro parser passado para `seq` falhar (retornar uma lista vazia de resultados)? E se o segundo parser falhar para algum sufixo da entrada?