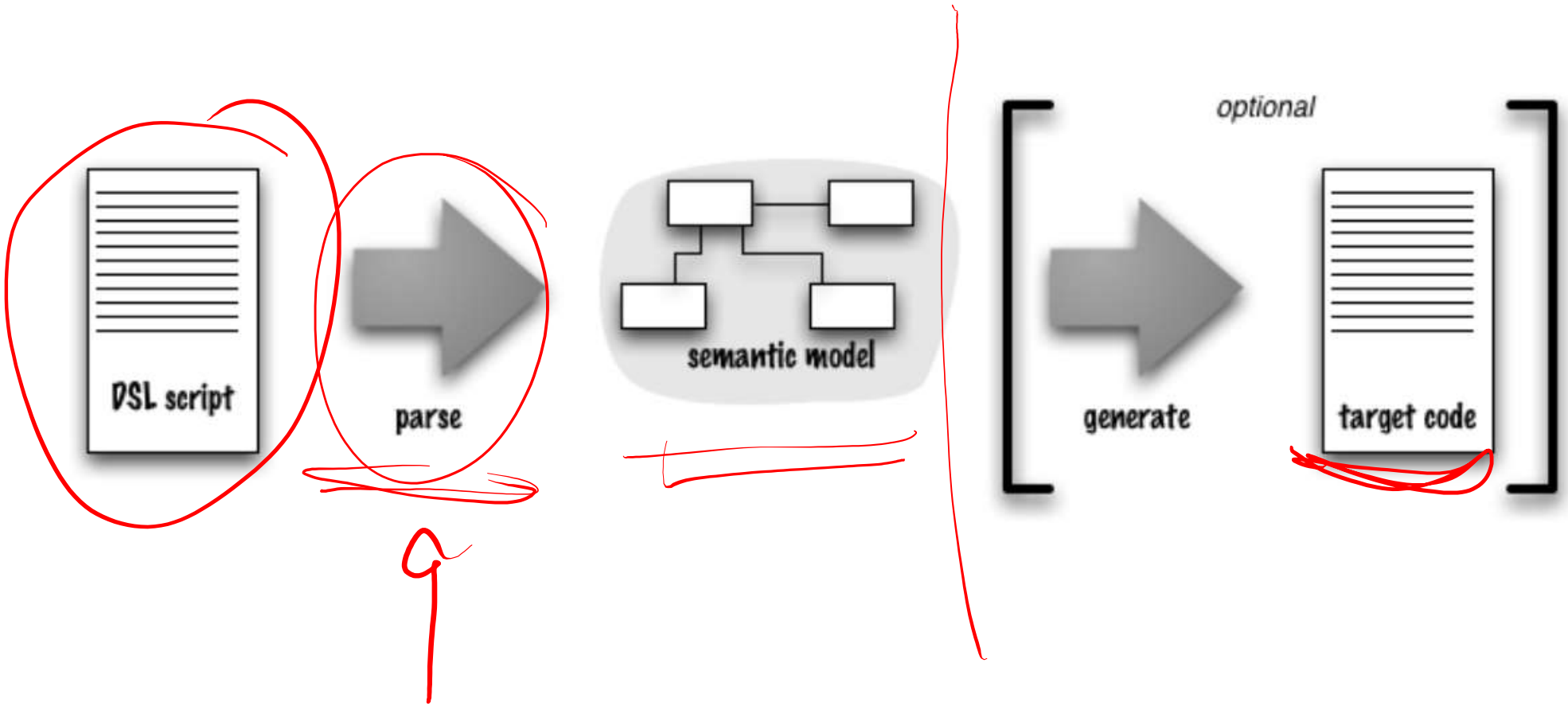
Linguagens de Domínio Específico

Fabio Mascarenhas – 2016.1

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

Processamento de uma DSL



Erros

- Uma falha em um parser de combinadores tem dois significados:
  - A alternativa que estamos tentando não está correta, mas outra estará
  - A entrada tem um erro de sintaxe
- A união dessas duas condições em um único estado do parser causa problemas para gerar boas mensagens de erro para o usuário!
- Um erro de sintaxe pode fazer o parser todo falhar sem indicar onde, ou fazer ele consumir apenas parte da entrada

Erros - exemplo

- Vamos ver o que acontece com a gramática a seguir:

```
bloco  -> stat*
stat   -> "while" exp "do" bloco "end" | NAME "=" exp
exp    -> aexp ">" aexp | aexp
aexp   -> termo (aop termo)*
termo  -> fator (mop fator)*
fator  -> NUM | NAME | "(" exp ")"
aop    -> "+" | "-"
mop    -> "*" | "/"
```

- Vamos analisar o programa abaixo, que tem um erro de sintaxe:

```
n = 5
f = 1
while n > 0 do
   f = f * n
   n n - 1        -- falta um =
end
```

Falha mais distante

- Uma estratégia para indicar ao usuário onde um erro aconteceu é guardar a posição na entrada onde aconteceu a falha mais distante
  - Ou, de maneira equivalente, o tamanho do menor sufixo da entrada onde uma falha aconteceu
- Isso requer que o parser mantenha mais esse estado: ao invés de receber a entrada e dar o resultado, ele recebe a entrada e o menor sufixo, e retorna o resultado e o menor sufixo (que pode ser outro)
- Todos os combinadores que manipulam diretamente a entrada precisam ser mudados

Falha mais distante - exemplo

- Vamos usar o mesmo exemplo de antes, e escrever uma função que faz o pós-processamento do menor sufixo e da entrada inicial para gerar uma mensagem de erro com a posição

Falha mais distante – terminais esperados

- Podemos melhorar ainda mais as mensagens de erro mantendo, além do menor sufixo, um conjunto de *terminais esperados*
- A ideia é incluir um terminal nesse conjunto toda vez que ele falhar com um sufixo de tamanho igual do do menor sufixo
- Na atualização do menor sufixo um novo conjunto é usado
- Ao final do parsing temos não apenas a posição onde o provável erro aconteceu, mas quais terminais (ou tokens) eram esperados naquela posição, ajudando o usuário a corrigir o erro

Combinadores “LL(1)”

- Uma outra maneira de detectar erros em parsers de combinadores é introduzir um valor de “erro” explícito, além da falha
- Esse valor interrompe a análise, diretamente no local onde o erro aconteceu
- Mas quando um parser deve sinalizar um erro?
- Uma ideia é assumir que uma restrição análoga à restrição LL(1) de gramáticas livres de contexto: no `seq`, se o primeiro parser consumiu algo da entrada, uma falha no parser subsequente é transformada em um erro
- LL(1) é bastante restrito, então também acrescentamos um jeito de aumentar o “lookahead”: um combinador `try` que transforma um erro que tenha acontecido em uma falha

Predicados sintáticos

- Um *predicado sintático* é um tipo de combinador que ou falha ou não consome nada da entrada
- O combinador `not` recebe um parser `p` e constrói um predicado que falha se `p` não falhar, e se `p` falhar simplesmente não consome nada
- O combinador `and` é o contrário de `not`, e falha caso `p` falhe e não consome nada caso `p` não falhe (qualquer coisa consumida por `p` é simplesmente ignorada)
- Predicados sintáticos permitem construir parsers que atuam diretamente nos caracteres da entrada, misturando análise léxica e sintática

PEGs

- As gramáticas de expressões de parsing, ou PEGs (parsing expression grammars) são uma linguagem para especificar parsers determinísticos

- Ao contrário das gramáticas livres de contexto, PEGs têm um mapeamento natural para os combinadores que estamos usando

- Uma *expressão de parsing* pode ser a expressão vazia `''`, um *terminal* `'a'`, um *não-terminal* `A`, uma *sequência* `pq`, onde `p` e `q` são expressões de parsing, uma *escolha ordenada* `p/q`, uma *repetição* `p*`, ou um *predicado* `!p`

- Uma PEG é simplesmente um mapeamento entre nomes de não-terminais e suas expressões de parsing

Árvore Sintática e AST

- A estrutura gramatical de uma entrada forma naturalmente uma árvore: tokens são folhas, outros termos sintáticos são nós internos

- Uma árvore sintática tem muita informação redundante, e vimos também que a gramática mais natural para uma linguagem nem sempre é a melhor para escrever um analisador para ela

- Resolvemos os dois problemas com uma *árvore sintática abstrata*: toda informação redundante é descartada, e procuramos a representação ideal da estrutura gramatical daquela linguagem, independentemente da gramática concreta que usamos