Compiladores - Análise Recursiva

Fabio Mascarenhas – 2015.2

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

Geradores x Reconhecedores

- A definição formal de gramática dá um *gerador* para uma linguagem
- Para análise sintática, precisamos de um *reconhecedor*
- Mas podemos reformular a definição de gramática para dar um reconhecedor, também
- Uma PE-CFG (gramática livre de contexto com expressões de parsing) tem os mesmos conjuntos V, T e P de uma gramática tradicional, mas o conjunto P é uma *função* de não-terminais em *expressões de parsing*
- Podemos ter ou um não-terminal inicial *S* ou uma expressão de pasing inicial *s*

Expressões de Parsing

- Uma expressão de parsing é:
  - Um terminal *a*
  - Um não-terminal *A*
  - Uma *concatenação* de duas expressões *pq*
  - Uma *escolha* entre duas expressões *p*|*q*

- A precedência da concatenação é maior que a da escolha, mas podemos usar parênteses para agrupamento

Reconhecendo uma entrada

- O significado de uma expressão de parsing *p* associada a uma gramática *G*, dada uma entrada qualquer, é dado por uma série de *regras de dedução* que dizem se a expressão *reconhece* um prefixo da entrada

$$\frac{}{G\ a\ ax \to x} \qquad \frac{G\ P(A)\ xy \to y}{G\ A\ xy \to y}$$

$$\frac{G\ p\ xyz \to yz \quad G\ q\ yz \to z}{G\ pq\ xyz \to z} \qquad \frac{}{G\ \varepsilon\ x \to x}$$

$$\frac{G\ p\ xy \to y}{G\ p|q\ xy \to y}\ \text{esc.1} \qquad \frac{G\ q\ xy \to y}{G\ p|q\ xy \to y}\ \text{esc.2}$$

Exemplo

$$\begin{cases} E \to T\{+T|-T\} \\ T \to n|(E) \end{cases}$$

$E \to T E'$

$E' \to (+ T E' | - T E' | \varepsilon)$

$T \to n | (E)$

$$\dfrac{\dfrac{\dfrac{\dfrac{G\,(\,(n+n)\to n+n)\,\cdots}{G\,(E)\,(n+n)\to\varepsilon}}{G\,n|(E)\,(n+n)\to\varepsilon}}{G\,T\,(n+n)\to\varepsilon}}{G\,+\,(n+n)\to(n+n)}$$

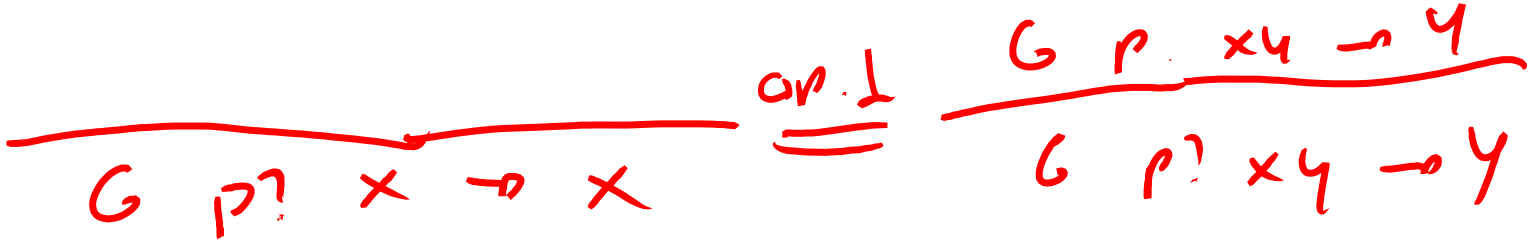
$$\dfrac{\dfrac{\dfrac{G\,n\;\; n+(n+n)\to+(n+n)}{G\,n|(E)\;\; n+(n+n)\to+(n+n)}}{G\,T\;\; n+(n+n)\to+(n+n)}}{G\,TE'\;\; n+(n+n)\to\varepsilon}$$

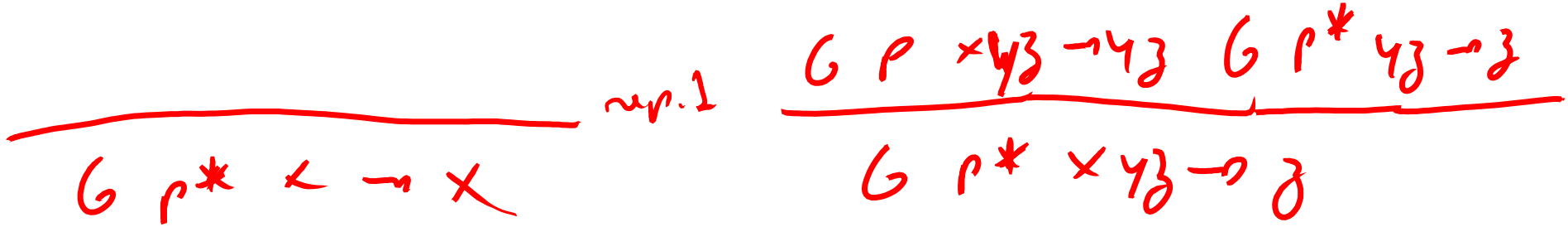
$$\dfrac{\dfrac{G\,+TE'\,+(n+n)\to\varepsilon}{G\,+TE'|-TE'|\varepsilon\;\to\varepsilon}}{G\,E'\,+(n+n)\to\varepsilon}$$

$$G\;E\;\;n+(n+n)\to\varepsilon$$

Repetição e Opcional

- Regras de dedução podem para repetição e opcional podem ser dadas diretamente:





Não-determinismo da escolha

- As regras de dedução para a escolha não dizem qual das alternativas escolher: a escolha em uma gramátia livre de contexto é *não-determinística*
- Simular não-determinismo em uma implementação real não é difícil, mas não é muito eficiente, e gera problemas de *ambiguidade*
- Todas as técnicas de análise sintática que vamos ver são diferentes maneiras de domar esse não-determinismo
- A primeira técnica, que vamos ver a seguir, reinterpreta a escolha para ser *determinística* e *ordenada*

* extern

Escolha ordenada

$$* \ \frac{a \neq b}{G \ a \ b \ x \rightarrow falha}$$

$$* \ \frac{G \ P(p) < \rightarrow falha}{G \ A \ x \rightarrow falha}$$

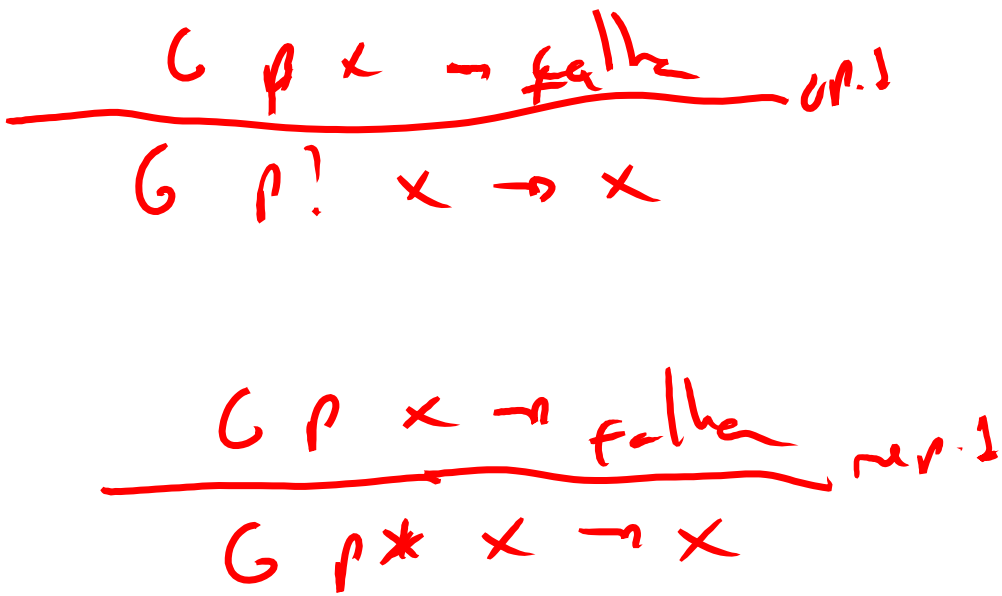
$$* \ \frac{G \ p \ x \rightarrow falha}{G \ p_{\S} \ x \rightarrow falha}$$

$$* \ \frac{G \ p \ x_1 \rightarrow x \quad G \ q \ x_2 \rightarrow falha}{G \ p_{\S} \ x_1 \rightarrow falha}$$

$$\frac{G \ p \ x_1 \rightarrow falha \quad G \ q \ x_1 \rightarrow y}{G \ p|_{\S} \ x_1 \rightarrow y} \ esc.2$$

$$* \ \frac{G \ p \ x \rightarrow falha \quad G \ q \ x \rightarrow falha}{G \ p|_{\S} \ x \rightarrow falha}$$

Escolha ordenada (2)



Analisador Recursivo

- Maneira mais simples de implementar um analisador sintático a partir de uma gramática, mas não funciona com muitas gramáticas
- A ideia é manter a lista de tokens em um vetor, e o token atual é um índice nesse vetor
- Um **terminal** testa o token atual, e avança para o próximo token se o tipo for compatível, ou falha se não for
- Uma **sequência** testa cada termo da sequência, falhando caso qualquer um deles falhe
- Uma **alternativa** guarda o índice atual e testa a primeira opção, caso falhe volta para o índice guardado e testa a segunda, assim por diante

Analisador Recursivo

- Um **opcional** guarda o índice atual, e testa o seu termo, caso ele falhe volta para o índice guardado e não faz nada
- Uma **repetição** repete os seguintes passos até o seu termo falhar: guarda o índice atual e testa o seu termo
- Um **não-terminal** vira um procedimento separado, e executa o procedimento correspondente
- Construir a árvore sintática é um pouco mais complicado, as alternativas, opcionais e repetições devem jogar fora nós da parte que falhou!

Retrocesso Local

- Podemos definir o processo de construção de um parser recursivo com retrocesso local como uma transformação de EBNF para código Java

- Os parâmetros para nossa transformação são o termo EBNF que queremos transformar e um termo Java que nos dá o objeto da árvore sintática

- Vamos chamar nossa transformação de `$parser`

- `$parser(termo, arvore)` dá o código para análise sintática do termo, guardando o resultado em um ou mais nós de `arvore` caso seja bem sucedido

Retrocesso Local

- `$parser(termo, arvore)` dá o código para análise sintática do termo, guardando o resultado em um ou mais nós de `arvore` caso seja bem sucedido

```
$parser(terminal, arvore) =
  ($arvore).child(match($terminal));

$parser(t1...tn, arvore) =
  $parser(t1, arvore)
  ...
  $parser(tn, arvore)

$parser(NAOTERM, arvore) =
  ($arvore).child(NAOTERM());
```

Retrocesso Local

- `$parser(termo, arvore)` dá o código para análise sintática do termo, guardando o resultado em um ou mais nós de `arvore` caso seja bem sucedido

```
$parser(t1 | t2, arvore) =
  {
    int atual = pos;
    try {
      Tree rascunho = new Tree();
      $parser(t1, rascunho);
      ($arvore).children.addAll(rascunho.children);
    } catch(Falha f) {
      pos = atual;
      $parser(t2, arvore);
    }
  }
```

Retrocesso Local

- `$parser(termo, arvore)` dá o código para análise sintática do termo, guardando o resultado em um ou mais nós de `arvore` caso seja bem sucedido

```
$parser([ termo ], arvore) =
  {
    int atual = pos;
    try {
      Tree rascunho = new Tree();
      $parser(termo, rascunho);
      ($arvore).children.addAll(rascunho.children);
    } catch(Falha f) {
      pos = atual;
    }
  }
```

Retrocesso Local

- `$parser(termo, arvore)` dá o código para análise sintática do termo, guardando o resultado em um ou mais nós de `arvore` caso seja bem sucedido

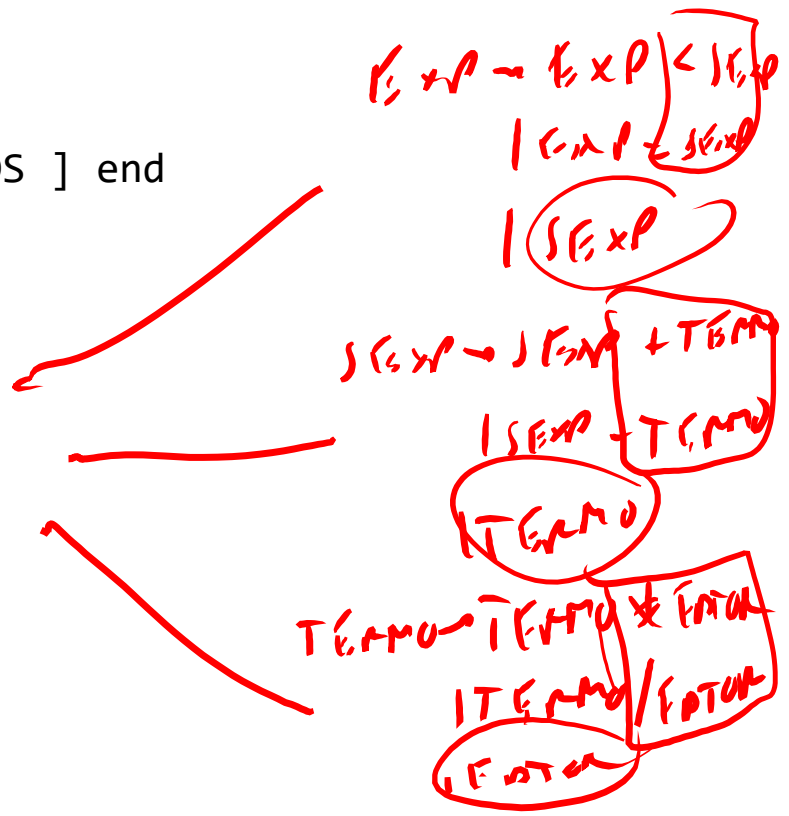
```
$parser({ termo }, arvore) =
  while(true) {
     int atual = pos;
     try {
        Tree rascunho = new Tree();
        $parser(termo, rascunho);
        ($arvore).children.addAll(rascunho.children);
     } catch(Falha f) {
        pos = atual;
        break;
     }
  }
```

Um analisador recursivo para TINY

- Vamos construir um analisador recursivo para TINY de maneira sistemática, gerando uma árvore sintática

- O vetor de tokens vai ser gerado a partir de um analisador léxico escrito com o JFlex

```
S      -> CMDS
CMDS   -> CMD { ; CMD }
CMD    -> if EXP then CMDS [ else CMDS ] end
        | repeat CMDS until EXP
        | id := EXP
        | read id
        | write EXP
EXP    -> SEXP { < SEXP | = SEXP }
SEXP   -> TERMO { + TERMO | - TERMO }
TERMO -> FATOR { * FATOR | / FATOR }
FATOR -> "(" EXP ")" | num | id
```



Detecção de erros

- Um analisador recursivo com retrocesso tem um comportamento ruim na presença de erros sintáticos

- Ele não consegue distinguir *falhas* (um sinal de que ele tem que tentar outra possibilidade) de *erros* (o programa está sintaticamente incorreto)

- Uma heurística é manter em uma variável global uma marca d’água que indica o quão longe fomos na sequência de tokens

Retrocesso local x global

- O retrocesso em caso de falha do nosso analisador é *local*. Isso quer dizer que se eu tiver `( A | B ) C` e A não falha mas depois C falha, ele não tenta B depois C novamente

- Da mesma forma, se eu tenho `A | A B` a segunda alternativa nunca vai ser bem sucedida

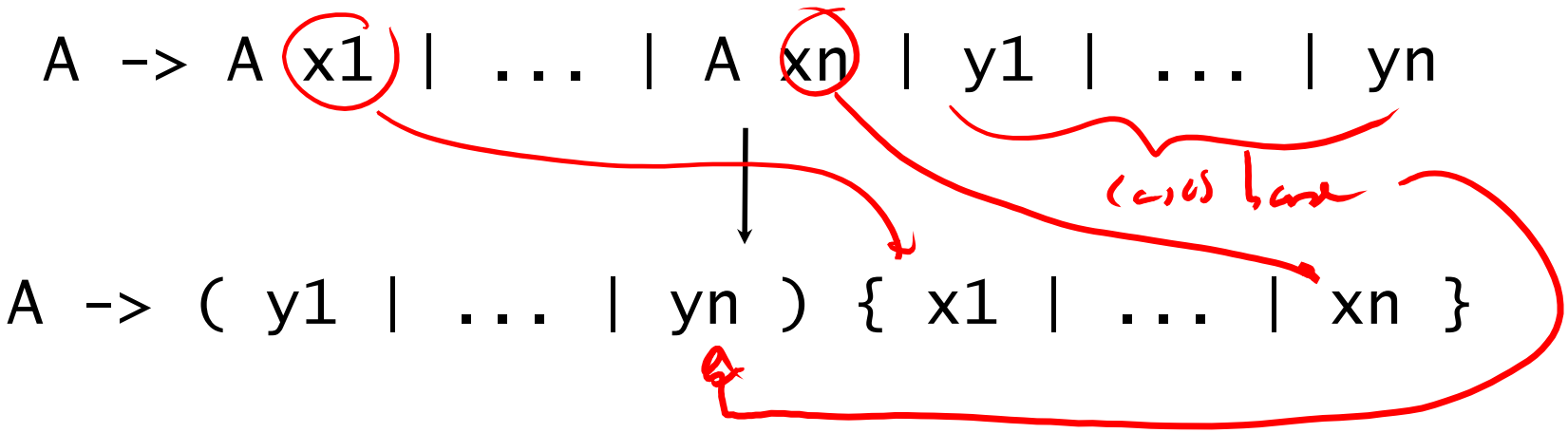
- As alternativas precisam ser *exclusivas*

- Retrocesso local também faz a repetição ser *gulosa*

- Uma implementação com retrocesso *global* é possível, mas mais complicada

Recursão à esquerda

- Outra grande limitação dos analisadores recursivos é que as suas gramáticas não podem ter *recursão à esquerda*
- A presença de recursão à esquerda faz o analisador entrar em um laço infinito!
- Precisamos transformar recursão à esquerda em repetição
- Fácil quando a recursão é direta:



Eliminação de recursão sem EBNF

```
A -> A x1
...
A -> A xn
A -> y1
...
A -> yn
```

```
A  -> y1 A'
...
A  -> yn A'
A' -> x1 A'
...
A' -> xn A'
A' ->
```

Parsing Expression Grammars

- As *parsing expression grammars* (PEGs) são uma generalização do parser com retrocesso local
- A sintaxe das gramáticas adota algumas características de expressões regulares: * e + para repetição ao invés de {}, ? para opcional ao invés de []
- Usa-se / para alternativas ao invés de |, para enfatizar que esse é um operador bem diferente do das gramáticas livres de contexto
- Acrescentam-se dois operadores de *lookahead*: *&e* e *!e*
- Finalmente, uma PEG pode misturar a tokenização com a análise sintática, então os terminais são *caracteres* (com sintaxe para strings e classes)

Uma PEG para TINY

```
S      <- CMDS
CMDS   <- CMD (; CMD)*
CMD    <- if EXP then CMDS (else CMDS)? end
        / repeat CMDS until EXP
        / id := EXP
        / read id
        / write EXP
EXP    <- SEXP (< SEXP / = SEXP)*
SEXP   <- TERMO (+ TERMO / - TERMO)*
TERMO  <- FATOR ("*" FATOR / "/" FATOR)*
FATOR  <- "(" EXP ")" / id / num
```