Compiladores - Análise Recursiva

Fabio Mascarenhas – 2018.1

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 parsing 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

$$\frac{}{G\ a\ ax \to (a, x)}$$

$$\frac{G\ p\ xy \to (w, y)}{G\ p|q\ xy \to (w, y)}$$

$$\frac{G\ G.p(\Lambda)\ xy \to (w, y)}{G\ A\ xy \to (\underset{w}{A}, y)}$$

$$\frac{G\ q\ xy \to (w, y)}{G\ p|s\ xy \to (w, y)}$$

$$\frac{G\ p\ xyz \to (w_1, yz)\quad G\ q\ yz \to (w_2, z)}{G\ pq\ xyz \to (w_1 w_2, z)}$$

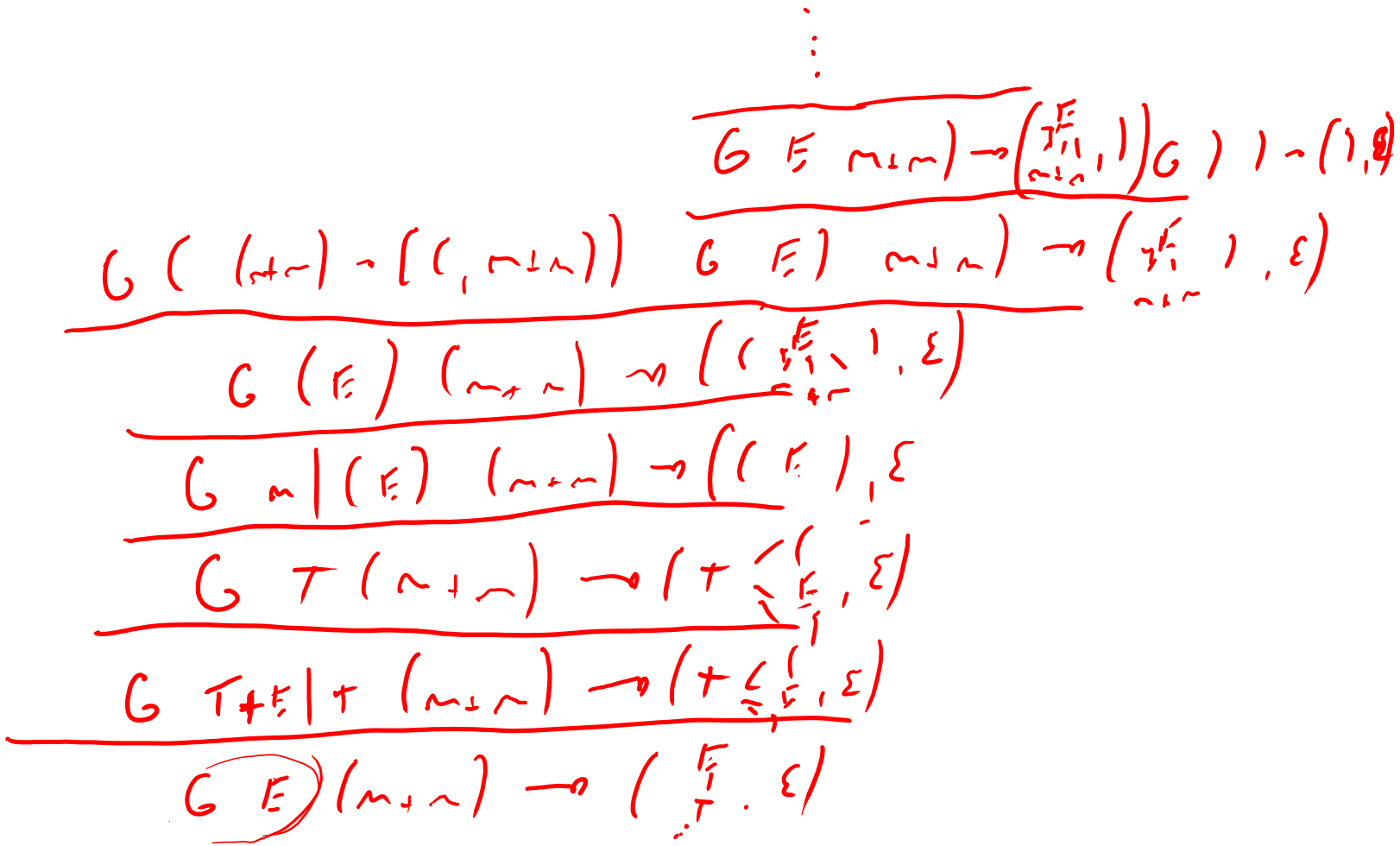
$$L(G) = \{ x \mid G\ G.s\ x \to (w, \varepsilon) \}$$

Exemplo

$E \to T + E \mid T$

$T \to n \mid (E)$

$E \to T \to (E) \to (T + E) \to (n + E) \to (n + T) \to (n + n)$



Não-determinismo da escolha

- As regras de dedução para a escolha não dizem qual das alternativas escolher: a escolha em uma gramática 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*

Escolha ordenada

$$\frac{G \; a \; b \; x \to \bot}{G \; a \; \varepsilon \to \bot}$$

$$\frac{G \; c.P(A) \; x \to \bot}{G \; \Box A \; x \to \bot}$$

$$\frac{G \; P \; x \to \bot}{G \; P \mid Q \; x \to \bot}$$

$$\frac{G \; P \; xy \to (w,y) \quad G \; Q \; y \to \bot}{G \; P \mid Q \; xy \to \bot}$$

substitui P por $P \mid Q$

$$\frac{G \; P \; xy \to \bot \quad G \; \S \; xy \to (w,y)}{G \; P \mid Q \; xy \to (w,y)}$$

$$\frac{G \; P \; x \to \bot \quad G \; \S \; x \to \bot}{G \; P \mid Q \; x \to \bot}$$

Analisador Recursivo (1)

- Maneira mais simples de implementar um analisador sintático a partir de uma gramática, mas não funciona com todas as 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 (2)

- Um **opcional** guarda o índice atual, e testa o seu termo, caso ele falhe volta para o índice guardado e não faz nada

$$(p) \equiv p \mid \varepsilon$$

- Uma **repetição** repete os seguintes passos até o seu termo falhar: guarda o índice atual e testa o seu termo

$$\{p\} \equiv pp^* \mid \varepsilon$$

- 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!

Construção do Analisador

- 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

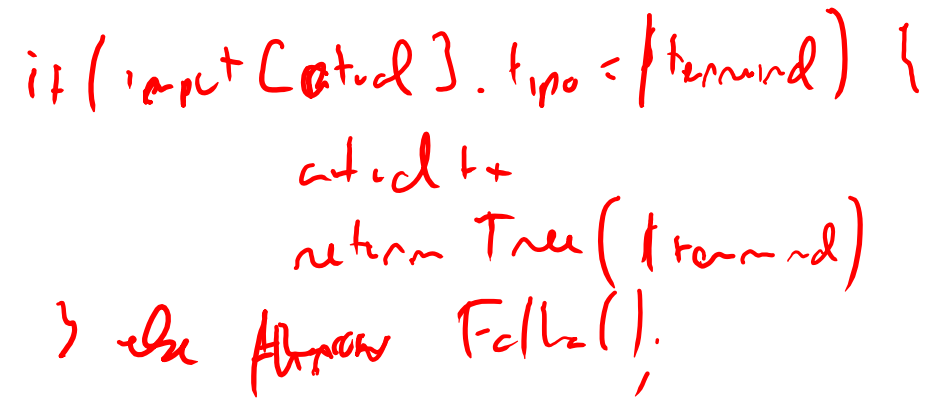
Regras de Construção (1)

- $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());
```



Regras de Construção (2)

- $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];
  }
 }
```

Regras de Construção (3)

- $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;
  }
 }
```

Regras de Construção (4)

- $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
```