

Compiladores - Análise Preditiva

Fabio Mascarenhas - 2013.2

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

Analisador Preditivo

- Uma simplificação do parser recursivo com retrocesso que é possível para muitas gramáticas são os *parsers preditivos*
- Um parser preditivo não tenta alternativas até uma ser bem sucedida, mas usa um *lookahead* na entrada para *prever* qual alternativa ele deve seguir
 - Só falha se realmente o programa está errado!
- Quanto mais tokens à frente podemos examinar, mais poderoso o parser
- Classe de gramáticas LL(k), onde k é quantos tokens de lookahead são necessários

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

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

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

S -> CMDS

CMDS -> CMD { ; CMD } 1

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

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

S -> CMDS

CMDS -> CMD { ; CMD } **1**

CMD -> if EXP then CMDS [else CMDS] **1** end
 | repeat CMDS until EXP
 | id := EXP
 | read id
 | write EXP

EXP -> SEXP ~~<~~ **1** < SEXP | = SEXP **1**

SEXP -> TERMO { + TERMO | - TERMO }

TERMO -> FATOR { * FATOR | / FATOR }

FATOR -> "(" EXP ")" | num | id

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

S -> CMDS

CMDS -> CMD { ; CMD } **1**

CMD -> if EXP then CMDS [else CMDS] **1** end
 | repeat CMDS until EXP
1 | id := EXP | ~~id(1) EXP 1 := EXP~~
1 | read id
 | write EXP

EXP -> SEXP **1** < SEXP **1** = SEXP **1**

SEXP -> TERMO { + TERMO | - TERMO }

TERMO -> FATOR { * FATOR | / FATOR }

FATOR -> "(" EXP ")" | num | id

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

S -> CMDS

CMDS -> CMD { ; CMD } **1**

CMD -> if EXP then CMDS [else CMDS] **1** end
 | repeat CMDS until EXP
1 | id := EXP
1 | read id
 | write EXP

EXP -> SEXP < SEXP | = SEXP **1**

SEXP -> TERMO { + TERMO | - TERMO }

TERMO -> FATOR { * FATOR | / FATOR }

FATOR -> "(" EXP ")" | num | id

The diagram illustrates the lookahead analysis for the EXP production. Red annotations highlight specific symbols that must be lookahead to correctly predict the grammar rules. A red bracket groups the alternatives for EXP: '< SEXP | = SEXP'. Red circles highlight the opening brace '{' before TERMO in SEXP, the closing brace '}' after TERMO in SEXP, the opening brace '{' before FATOR in TERMO, and the closing brace '}' after FATOR in TERMO.

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

S -> CMDS

CMDS -> CMD { ; CMD } **1**

CMD -> if EXP then CMDS [else CMDS] **1** end
 | repeat CMDS until EXP

1 | id := EXP

1 | read id

1 | write EXP

EXP -> SEXP < SEXP | = SEXP > **1**

SEXP -> TERMO { + TERMO | - TERMO } **1**

TERMO -> FATOR { * FATOR | / FATOR } **1**

FATOR -> "(" EXP ")" | num | id

The diagram illustrates the lookahead requirements for each grammar rule. Red circles highlight specific lookahead symbols: a left parenthesis for EXP, a plus sign for SEXP, a minus sign for TERMO, a multiplication sign for FATOR, and a division sign for FATOR. Red arrows point from these circled symbols to their respective positions in the grammar rules.

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

S -> CMDS

CMDS -> CMD { ; CMD } **1**

CMD -> if EXP then CMDS [else CMDS] **1** end
 | repeat CMDS until EXP
 1 | id := EXP
 1 | read id
 1 | write EXP

EXP -> SEXP [< SEXP | = SEXP] **1**

SEXP -> TERMO { + TERMO | - TERMO } **1**

TERMO -> FATOR { * FATOR | / FATOR } **1**

FATOR -> "(" EXP ")" | num | id

1

Voltando a TINY

- Vamos ver quantos tokens de lookahead precisamos para prever cada opção:

S → CMDS

CMDS → CMD { ; CMD } **1**

CMD → if EXP then CMDS [else CMDS] **1** end
| repeat CMDS until EXP

1 | id := EXP

| read id

| write EXP

EXP → SEXP (< SEXP | = SEXP) **1**

SEXP → TERMO { + TERMO | - TERMO } **1**

TERMO → FATOR { * FATOR | / FATOR } **1**

FATOR → "(" EXP ")" | num | id

1

TINY é LL(1)!

Analisador preditivo para TINY

- O analisador ~~recursivo~~ preditivo é bem mais simples do que o analisador com retrocesso
- Pode ler os tokens sob demanda: só precisa manter um token de lookahead
- Não precisamos de nada especial para detecção de erros: os pontos de falha são pontos de erro, e temos toda a informação necessária lá
- Temos os mesmos problemas com recursão à esquerda

MAS PODEMOS USAR UM TAV QUE
P/ CONSTRUIR A ÁRVORE COMO
SE A GRAMÁTICA FOSSE R. E.

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:

$$\begin{array}{l} A \rightarrow \textcircled{A} \ x_1 \ | \ \overbrace{\dots}^{\equiv} \ | \ A \ x_n \ | \ y_1 \ | \ \dots \ | \ y_n \\ \downarrow \\ A \rightarrow (y_1 \ | \ \dots \ | \ y_n) \ \{ \ x_1 \ | \ \dots \ | \ x_n \ } \end{array}$$

tree p() {
tree t; new('n');
t.child(p());
}

Eliminação de recursão sem EBNF

