Compiladores - Análise Preditiva

Fabio Mascarenhas – 2015.2

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 }
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 }
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
```

1

1

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
```

1

1

1

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
```

1

1

1

1

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
```

1 1 1 1 1 1

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
```

1 1 1 1 1 1 1

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
```

1 1 1 1 1 1

TINY é LL(1)!

Analisador Recursivo Preditivo

- O analisador recursivo preditivo é parecido com o recursivo com retrocesso, mas ao invés de *try* nas escolhas usa a informação de lookahead
- Um **terminal** continua testando o token atual, e avançando para o próximo token se o tipo for compatível, mas lança um ***erro*** se não for
- Uma **sequência** simplesmente executa cada termo da sequência
- Uma **alternativa** usa o token de lookahead como índice de um *switch-case*, e o conjunto de lookahead de cada alternativa forma os casos dela;

Analisador Recursivo Preditivo

- Um **opcional** verifica se o token de lookahead está no conjunto do seu termo, caso esteja segue executando ele, senão pula

- Uma **repetição** repete os seguintes passos: verifica se o token de lookahead está no conjunto de seu termo, se estiver executa ele, se não pára a repetição

- Um **não-terminal** vira um procedimento separado, e executa o procedimento correspondente

- Construir a árvore sintática é simples, já que não há retrocesso, e pode-se sempre adicionar nós à arvore atual

Recursivo Preditivo

- `$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 Preditivo

- `$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) =
  {
    switch(lookahead.tipo) {
      // { c1, ..., cn } é o conjunto de lookahead de t1
      case 'c1': ... case 'cn':
        $parser(t1, arvore);
      default:
        $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) =
  switch(lookahead.tipo) {
    // { c1, ..., cn } é o conjunto de lookahead de termo
    case ‘c1’: ... case ‘cn’:
      $parser(termo, arvore);
  }

$parser({ termo }, arvore) =
  // { c1, ..., cn } é o conjunto de lookahead de termo
  while(lookahead.tipo == ‘c1’ || ... || lookahead.tipo == ‘cn’) {
    $parser(termo, arvore);
  }
```

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

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:

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

```
A -> ( y1 | ... | yn ) { x1 | ... | xn }
```

Eliminação de recursão sem EBNF