

Compiladores - Análise Preditiva

Fabio Mascarenhas - 2013.1

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

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(terminal, 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;  
    }  
  }  
}
```

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

Detecção de erros

- Um analisador recursivo com retrocesso também 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

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:

$$A \rightarrow A x_1 \mid \dots \mid A x_n \mid y_1 \mid \dots \mid y_n$$

$$A \rightarrow (y_1 \mid \dots \mid y_n) \{ x_1 \mid \dots \mid x_n \}$$

Eliminação de recursão sem EBNF

$A \rightarrow A x_1$

...

$A \rightarrow A x_n$

$A \rightarrow y_1$

...

$A \rightarrow y_n$



$A \rightarrow y_1 A'$

...

$A \rightarrow y_n A'$

$A' \rightarrow x_1 A'$

...

$A' \rightarrow x_n A'$

$A' \rightarrow$

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*: &t e !t
- Finalmente, uma PEG pode misturar a tokenização com a análise sintática, então os terminais são *caracteres* (com sintaxe para strings e classes)

Parsers preditivos

- 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 COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
        | id := EXP
        | read id
        | write EXP
COND   -> EXP ( < EXP | = EXP )
EXP    -> 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 1
CMDS   -> CMD { ; CMD }
CMD    -> if COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
        | id := EXP
        | read id
        | write EXP
COND   -> EXP ( < EXP | = EXP )
EXP    -> 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 1
CMDS   -> CMD { ; CMD } 1
CMD    -> if COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
        | id := EXP
        | read id
        | write EXP
COND   -> EXP ( < EXP | = EXP )
EXP    -> 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 1
CMDS   -> CMD { ; CMD } 1
CMD    -> if COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
1    | id := EXP
        | read id
        | write EXP
COND   -> EXP ( < EXP | = EXP )
EXP    -> 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 1
CMDS   -> CMD { ; CMD } 1
CMD    -> if COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
1    | id := EXP
        | read id
        | write EXP 1
COND   -> EXP ( < EXP | = EXP )
EXP    -> 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 1
CMDS   -> CMD { ; CMD } 1
CMD    -> if COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
1    | id := EXP
        | read id
        | write EXP 1
COND   -> EXP ( < EXP | = EXP ) 1
EXP    -> TERMO { + TERMO | - TERMO } 1
TERMO  -> FATOR { * FATOR | / FATOR } 1
FATOR  -> "(" EXP ")" | num | id
```

Voltando a TINY

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

```
S      -> CMDS 1
CMDS   -> CMD { ; CMD } 1
CMD    -> if COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
1    | id := EXP
        | read id
        | write EXP
COND   -> EXP ( < EXP | = EXP ) 1
EXP    -> 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 1
CMDS   -> CMD { ; CMD } 1
CMD    -> if COND then CMDS [ else CMDS ] end
        | repeat CMDS until COND
1    | id := EXP
        | read id
        | write EXP
COND   -> EXP ( < EXP | = EXP ) 1
EXP    -> TERMO { + TERMO | - TERMO } 1
TERMO  -> FATOR { * FATOR | / FATOR } 1
FATOR  -> "(" EXP ")" | num | id
```

1

TINY é LL(1)!

Analizador preditivo para TINY

- O analisador recursivo preditivo é bem mais simples do que o analisador com retrocesso
- Pode ler os tokens sob demanda, também, só precisa manter um token de *lookahead*