Compiladores - Análise LL

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(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;
     }
  }
```

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

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

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*: &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 }
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 }
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                                1
COND   -> EXP ( < EXP | = EXP )                          1
EXP    -> TERMO { + TERMO | - TERMO }                    1
TERMO  -> FATOR { * FATOR | / FATOR }
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, também, só precisa manter um token de *lookahead*

Gramáticas LL(1)

- Uma gramática é LL(1) se toda predição pode ser feita examinando um único token à frente
- Muitas construções em gramáticas de linguagens de programação são LL(1), ou podem ser tornadas LL(1) com alguns “jeitinhos”
- A vantagem é que um analisador LL(1) é bastante fácil de construir, e muito eficiente
- Como a análise LL(1) funciona?

Análise LL(1)

- Conceitualmente, o analisador LL(1) constrói uma *derivação mais à esquerda* para o programa, partindo do símbolo inicial
- A cada passo da derivação, o prefixo de terminais da forma sentencial tem que casar com um prefixo da entrada
- Caso exista mais de uma regra para o não-terminal que vai gerar o próximo passo da derivação, o analisador usa o primeiro token após esse prefixo para escolher qual regra usar
- Esse processo continua até todo o programa ser derivado ou acontecer um erro (o prefixo de terminais da forma sentencial não casa com um prefixo do programa)

Exemplo

- Uma gramática LL(1) simples:

```
PROG -> CMD ; PROG
PROG ->
CMD  -> id = EXP
CMD  -> print EXP
EXP  -> id
EXP  -> num
EXP  -> ( EXP + EXP )
```

- Vamos analisar `id = ( num + id ) ; print num ;`

Exemplo

- O terminal entre || é o *lookahead*, usado para escolher qual regra usar

```
PROG

|id| = ( num + id ) ; print num ;
```

Exemplo

- O terminal entre || é o *lookahead*, usado para escolher qual regra usar

```
PROG -> CMD ; PROG

|id| = ( num + id ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

```
PROG -> CMD ; PROG -> id = EXP ; PROG

id = |(| num + id ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

```
PROG -> CMD ; PROG -> id = EXP ; PROG ->
   id = ( EXP + EXP ) ; PROG

id = ( |num| + id ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

```
PROG -> CMD ; PROG -> id = EXP ; PROG ->
   id = ( EXP + EXP ) ; PROG -> id = ( num + EXP ) ; PROG

id = ( num + |id| ) ; print num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

```
PROG -> CMD ; PROG -> id = EXP ; PROG ->
   id = ( EXP + EXP ) ; PROG -> id = ( num + EXP ) ; PROG ->
   id = ( num + id ) ; PROG
```

```
id = ( num + id ) ; |print| num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

```
PROG -> CMD ; PROG -> id = EXP ; PROG ->
   id = ( EXP + EXP ) ; PROG -> id = ( num + EXP ) ; PROG ->
   id = ( num + id ) ; PROG -> id = ( num + id ) ; CMD ; PROG
```

```
id = ( num + id ) ; |print| num ;
```

Exemplo

- Um prefixo de terminais na forma sentencial desloca o lookahead

```
PROG -> CMD ; PROG -> id = EXP ; PROG ->
   id = ( EXP + EXP ) ; PROG -> id = ( num + EXP ) ; PROG ->
   id = ( num + id ) ; PROG -> id = ( num + id ) ; CMD ; PROG ->
   id = ( num + id ) ; print EXP ; PROG
```

```
id = ( num + id ) ; print |num| ;
```

Exemplo

- O lookahead agora está no final da entrada (EOF)

```
PROG -> CMD ; PROG -> id = EXP ; PROG ->
   id = ( EXP + EXP ) ; PROG -> id = ( num + EXP ) ; PROG ->
   id = ( num + id ) ; PROG -> id = ( num + id ) ; CMD ; PROG ->
   id = ( num + id ) ; print EXP ; PROG ->
   id = ( num + id ) ; print num ; PROG


id = ( num + id ) ; print num ; ||
```

Exemplo

- Chegamos em uma derivação para o programa, sucesso!

```
PROG -> CMD ; PROG -> id = EXP ; PROG ->
   id = ( EXP + EXP ) ; PROG -> id = ( num + EXP ) ; PROG ->
   id = ( num + id ) ; PROG -> id = ( num + id ) ; CMD ; PROG ->
   id = ( num + id ) ; print EXP ; PROG ->
   id = ( num + id ) ; print num ; PROG ->
   id = ( num + id ) ; print num ;


id = ( num + id ) ; print num ; ||
```

FIRST, FOLLOW e FIRST+

- Como o analisador LL(1) sabe qual regra aplicar, dado o lookahead?

- Examinando os *conjuntos de lookahead* (FIRST+) de cada regra

```
FIRST+(A -> w) = | FIRST(w) U FOLLOW(A) - { '' }, se '' em FIRST(w)
                 | FIRST(w), caso contrário
```

- E quem são os conjuntos FIRST e FOLLOW? Revisão de linguagens formais!

```
FIRST(w) = { x é terminal | w -*-> xv, para alguma string v } U
  { '' | w -*-> '' }
FOLLOW(A) = { x é terminal ou EOF | S -*-> wAxv para algum w e v }
```

A condição LL(1)

- Uma gramática é LL(1) se os conjuntos FIRST+ das regras de cada não-terminal são *disjuntos*

- Por que isso faz a análise LL(1) funcionar? Vejamos um passo LL(1):

```
...At... -*-> ...t...  ou
...A...  -*-> ...tw...
```

- No primeiro caso isso quer dizer que `t` está no FOLLOW(A)

- No segundo caso, `t` está no FIRST da regra de A que foi usada

- A derivação é mais à esquerda, então o primeiro ... é um prefixo de terminais, logo `t` é o lookahead!

Analisador LL(1) de tabela

- No analisador LL(1) recursivo, o contexto de análise (onde estamos na árvore sintática) é mantido pela pilha de chamadas da linguagem

- Mas podemos escrever um analisador LL(1) genérico (que funciona para qualquer gramática LL(1)), mantendo esse contexto em uma pilha explícita!