Compiladores

Análise sintática (2)
Análise Top-Down

Gramáticas Livres de Contexto Análise Top-Down

Gramática: $S \rightarrow A\ B$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow cbB \mid ca$

String: ccbca

Top-Down

$S \Rightarrow AB$	$S \rightarrow AB$
$\Rightarrow cB$	$A \rightarrow c$
$\Rightarrow ccbB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$

Transformações de GLCs

- Eliminação de produções vazias
- Eliminação da recursão à esquerda:
  - Recursão direta
    - A produção: $A \rightarrow Aa \mid b$
    - Se torna: $A \rightarrow bA'$
      $A' \rightarrow aA' \mid \varepsilon$
- Fatoração de uma gramática
  - A produção: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
  - Se torna: $A \rightarrow \alpha X$ e $X \rightarrow \beta_1 \mid \beta_2$

Plano da aula

- **Implementação de um reconhecedor de sentenças**
  - **Top-Down**
    - **Lookahead.**
    - **Primitivas First e Follow.**

Análise Top-Down

- Como implementar um reconhecedor para uma GLC?
- Constrói-se a árvore de derivação, lendo a sentença de esquerda para a direita, e substituindo sempre o não-terminal mais à esquerda.
- Existem três tipos principais de parser top-down:
  - Recursivo com retrocesso
  - Recursivo preditivo
  - Tabular preditivo.

Análise Top-Down – Exemplo

$S \rightarrow A\ B$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow cbB \mid ca$

- Gera $S \Rightarrow^* cbca$?

S	cbca	tentar $S \rightarrow AB$
AB	cbca	tentar $A \rightarrow c$
cB	cbca	casar c
B	bca	sem-saída, tentar $A \rightarrow \varepsilon$
εB	cbca	tentar $B \rightarrow cbB$
cbB	cbca	casar c
bB	bca	casar b
B	ca	tentar $B \rightarrow cbB$
cbB	ca	casar c
bB	a	sem-saída, tentar $B \rightarrow ca$
ca	ca	casar c
a	a	casar a -> Fim!

Análise Recursiva com Retrocesso

- Cria-se um procedimento por não-terminal
  - Testa-se aplicação de cada produção associada ao não-terminal;
  - Lê no texto de entrada o próximo token
    - Invoca o analisador lexical (yylex())
- É necessário lembrar onde se fez uma escolha de uma alternativa, para poder retroceder neste ponto.
-
- Operação de **retrocesso** (backtracking)

Análise Recursiva com Retrocesso

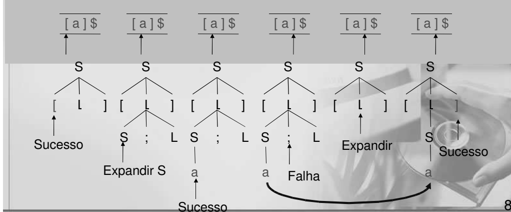
Exemplo (Price e Toscani, Seção 3.2.1)

Gramática:

S → a | [L]
L → S ; L | S

Entrada:

[a] $



Análise Recursiva com Retrocesso

```
token tok;

main() {
 tok = lexyy();
 if( S() )
  if( token == $ )
   printf("Sucesso!!!");
  else printf("Erro.");
}

boolean S() {
  if( tok == "a" ) {
   tok = lexyy();
   return true; }
  else if( tok == "]" ) {
   tok = lexyy();
   if( L() )
    if( tok = "]" ) { tok = lexyy(); return true; }
    else return false;
   else return false;
}
```

```
boolean L() {
  MARCA_PONTO;
  if( S() )
   if( tok == ";" ) {
    tok = lexyy();
    if( L() ) return true;
    else return false;
   }
   else {
    RETROCEDE; //o cabeçote de leitura
    if( S() ) return true;
    else return false;
   }
  else return false;
}
```


Análise Recursiva com Retrocesso

- Este método é ineficiente:
  - Várias leituras para a mesma entrada.
  - Difícil de de indicar local onde foi encontrado um erro, devido a tentativa de aplicação de produções alternativas.
  - Como em algumas situações o reconhecimento é acompanhado de ações semânticas que modificam a tabela de símbolos – retrocessos implicam em desfazer estas modificações.
- Não é um método comumente aplicado.

Análise Recursiva Preditiva

- Analisadores recursivos sem retrocesso.
- Chamados de preditivos pois o símbolo sob o cabeçote de leitura determina qual produção a ser aplicada a cada não terminal.
- Mais eficientes.

Análise Recursiva Preditiva

- É fácil de implementar.
- É necessário:
  1. Que a gramática não seja recursiva à esquerda
     - Pois: A → Aa
     - Implica na função: (recursão infinita)
       ```
       reconheceA() {
         reconheceA();
         ...
       }
       ```
  2. Que a gramática seja fatorada à esquerda
     - Senão deve se fazer retrocesso.
  3. Que os primeiros terminais deriváveis possibilitem a decisão de uma produção a aplicar!
     - Não há retrocesso sobre não-terminais...

Análise Recursiva Preditiva

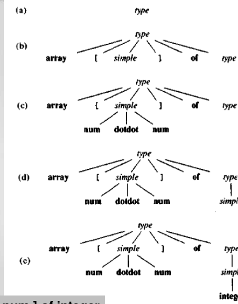
- Exemplo:

Aho, Seção 2.4

TYPE → *SIMPLE*
   | **^ id**
   | **array [** *SIMPLE* **] of** *TYPE*
SIMPLE → **integer**
   | **char**
   | **num dotdot num**

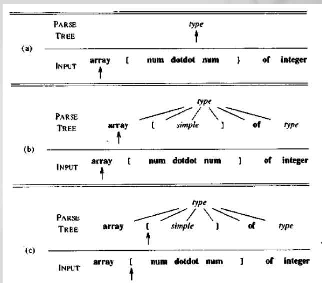
array [num dotdot num] of integer

Análise Recursiva Preditiva



array [num dotdot num] of integer

Análise Recursiva Preditiva



array [num dotdot num] of integer

Análise Recursiva Preditiva

```
token lookahead;
token nexttoken;
void match( token t ) {
 if( lookahead == t )
  lookahead = nexttoken;
 else error();
}
void type() {
 switch( lookahead )
  case integer :
  case char    :
  case num     : simple(); break;
  case  "^"    : match("^"); match( id ); break;
  case array   : match(array); match("[");
                 simple();
                 match("]"); match(of); type();
                 break;
  default : error();
 }
}
```

```
void simple() {
 switch( lookahead )
  case integer : match(integer);
                 break;
  case char    : match(char);
                 break;
  case num     : match(num);
                 match(dotdot);
                 match(num);
                 break;
  default: error();
 }
}
```


Analisador recursivo preditivo

- Conjunto First(α):
  - Definição formal:
    - Se existe um $t \in$ **T** e um $\beta \in$ **V*** tal que $\alpha \Rightarrow^* t\,\beta$ então $t \in \text{First}(\alpha)$
    - Se $\alpha \Rightarrow^* \varepsilon$ então $\varepsilon \in \text{First}(\alpha)$

A → B | C | D
B → b
C → c
D → d

First(A)={b,c,d}
First(B)={b}
First(C)={c}
First(D)={d}

  - Definição informal:
    - Conjunto de todos os terminais que começam qualquer seqüência derivável de α.

Analisador recursivo preditivo

- No caso que os First() são "simpáticos", não terá retrocesso.
- Isso supõe que para qualquer produção

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

se tenha

$$\text{First}(\alpha_1) \cap \text{First}(\alpha_2) \cap \dots \cap \text{First}(\alpha_n) = \varnothing$$

proc First(α: string de símbolos)

```
// seja α= X1 X2 X3 … Xn

if( X1 ∈ T ) // caso simples onde X1 é um terminal
  First(α) := {X1}
else {  // X1 não é um terminal
    i := 0;
    First(α) = First{X1} \ {ε};
    for( i = 1 ; i <= n ; i++ ) {
       if ε is in First(X1) and in First(X2) and in… First(Xi-1)
            First(α) := First(α) ∪ First(Xi) \ {ε}
    }
}
if( α→ ε ) // é uma produção
  then First(α) := First(α) ∪ {ε}
```


proc First(α: string de símbolos)

Exercício:

- Encontre o conjunto First para o seguinte conjunto de produções:

```
CMD → CONDICIONAL
    | INTERATIVO
    | ATRIBUICAO
CONDICIONAL  → if EXPR then CMD
ITERATIVO    → while EXPR do CMD
             | repeat LISTA until EXPR
ATRIBUICAO   → id := EXPR
```


O que acontece se ε ∈ First(A)?

S → ACE
A → a | b | ε
C → c | d | ε
E → e

Consegue derivar ?
- ace
- bde
- ce
- de
- e

O que acontece se ε ∈ First(A)?

S → ACE
A → a | b | ε
C → c | d | ε
E → e

```
S () {
switch {
case 'a':
case 'b':
     A();
     C();
     E();
     break;
default:
     abort("syntax_error");
}
```

Consegue derivar ?
- ace
- bde
- ce
- de
- e

O que acontece se ε ∈ First(A)?

S → ACE
A → a | b | ε
C → c | d | ε
E → e

```
S () {
switch {
case 'a':
case 'b':
case 'c':
case 'd':
     C();
     E();
     break;
default:
     abort("syntax_error");
}
```

Consegue derivar ?
- ace
- bde
- ce
- de
- e

O que acontece se ε ∈ First(A)?

S → ACE
A → a | b | ε
C → c | d | ε
E → e

```
S () {
switch {
case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
     E();
     break;
default:
     abort("syntax_error");
}
```

Consegue derivar ?
- ace
- bde
- ce
- de
- e

O que acontece se $\varepsilon \in$ First(A)?

$S \rightarrow ACE$
$A \rightarrow a \mid b \mid \varepsilon$
$C \rightarrow c \mid d \mid \varepsilon$
$E \rightarrow e$

```
A () {
switch {
case 'a':
    matchToken('a'); break;
case 'b':
    matchToken('b'); break;
case ??:
    break;
default:
    abort("syntax_error");
}
```

Consegue derivar ?
- ace
- bde
- ce
- de
- e

Exercício

Escrever um reconhecedor recursivo para:
G=({+,*,id,(,)} {E,T,F}, P, E)

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid$ **id**

Exercício

Escrever um reconhecedor recursivo para:
G=({+,*,id,(,)} {E,T,F}, P, E)

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid$ **id**

Retirar recursão à esquerda

Transformar produções do tipo
$A \rightarrow A\alpha_1 \mid A\alpha_2 \dots \mid \alpha_n \mid \beta_1 \mid \beta_2 \dots \mid \beta_m$

Em produções do tipo
$A \rightarrow \beta_1 X \mid \beta_2 X \dots \mid \beta_m X$

$X \rightarrow \alpha_1 X \mid \alpha_2 X \dots \mid \alpha_n X \mid \varepsilon$

Exercício

Escrever um reconhecedor recursivo para:
G=({+,*,id,(,)} {E,T,F}, P, E)

$E \rightarrow E+T \mid T$
A : E
α_1 : +T
β_1 : T

$T \rightarrow T*F \mid F$
A : T
α_1 : *F
β_1 : F

$F \rightarrow (E) \mid$ **id**

Retirar recursão à esquerda

Transformar produções do tipo
$A \rightarrow A\alpha_1 \mid A\alpha_2 \dots \mid \alpha_n \mid \beta_1 \mid \beta_2 \dots \mid \beta_m$

Em produções do tipo
$A \rightarrow \beta_1 X \mid \beta_2 X \dots \mid \beta_m X$

$X \rightarrow \alpha_1 X \mid \alpha_2 X \dots \mid \alpha_n X \mid \varepsilon$

Exercício

Escrever um reconhecedor recursivo para:
G=({+,*,id,(,)} {E,T,F}, P, E)

$E \rightarrow E+T \mid T$
A : E
α_1 : +T
β_1 : T

$T \rightarrow T*F \mid F$
A : T
α_1 : *F
β_1 : F

$F \rightarrow (E) \mid$ **id**

Retirar recursão à esquerda

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Para melhorar: o conjunto Follow

- Follow(A):
  - Conjunto de terminais que podem aparecer à direita de um não-terminal A em uma sentença válida.
  - $ passa a denotar um terminal "virtual" que marca o fim da entrada (EOF, CTRL-D,…)
- Formalmente:
  - Se existe um $t \in$ **T** e $\alpha, \beta \in$ **V*** tal que $S \Rightarrow^* \alpha A t \beta$ então $t \in$ Follow(A)
  - Se $S \Rightarrow^* \alpha A$ então $\$ \in$ Follow(A)

Exemplo First/Follow

$S \rightarrow A\ B$
$A \rightarrow c \mid \varepsilon$
$B \rightarrow cbB \mid ca$

First(A) = {c, ε} Follow(A) = {c}
First(B) = {c} Follow(B) = {$}
First(S) = {c} Follow(S) = {$}

Algoritmo: proc Follow(A ∈ N)

```
Follow(S) := {$};
Repeat
 foreach p ∈ P do {
  // Varre as produções
   case p == A → αBβ {
      Follow(B) := Follow(B) ∪ First(β)\{ε};
      if ε ∈ First(β) then
        Follow(B) := Follow(B) ∪ Follow(A);
      end
   }
   case p == A → αB
      Follow(B) := Follow(B) ∪ Follow(A);
} until no change in any Follow(N)
```


Exercício First/Follow

Para: G = (T,N,P,S)
T = {a,b,c,d,e,f}
N = {S,X,Y,Z}

P: $S \rightarrow XYZ$
$X \rightarrow aXb \mid \varepsilon$
$Y \rightarrow cYZcX \mid d$
$Z \rightarrow eZYe \mid f$

Exercício First/Follow

Para: G = (T,N,P,S)
T = {a,b,c,d,e,f}
N = {S,X,Y,Z}

P: $S \rightarrow XYZ$
$X \rightarrow aXb \mid \varepsilon$
$Y \rightarrow cYZcX \mid d$
$Z \rightarrow eZYe \mid f$

First(X) = {a, ε} Follow(X) = {c, d, b, e, f}
First(Y) = {c, d} Follow(Y) = {e, f}
First(Z) = {e, f} Follow(Z) = {$, c, d}
First(S) = {a, c, d} Follow(S) = {$}

Gramática LL(1)

Condições necessárias:
- Sem ambigüidade
- Sem recursão a esquerda

Uma gramática G é LL(1) sse
Para quaisquer duas produções de G
$A \rightarrow \alpha \mid \beta \Rightarrow^* t$

1. $First(\alpha) \cap First(\beta) = \emptyset$
2. $\alpha \Rightarrow^* \varepsilon$ implies $!(\beta \Rightarrow^* \varepsilon)$
3. $\alpha \Rightarrow^* \varepsilon$ implies $First(\beta) \cap Follow(A) = \emptyset$

(2) Significa que **ε** pertence no máximo ao First de um símbolo.

LL(1) = leitura Left -> right
+ derivação mais a esquerda (Left) +
+ uso de 1 token lookahead.

Sumário

- Gramáticas LL(1) podem ser analisadas por um simples parser descentente recursivo
  - Sem recursão a esquerda
  - Fatorada a esquerda
  - 1 símbolo de look-ahead
- Nem todas as linguagens podem ser tornadas LL(1).
- Ainda se pode usar um mecanismo mais poderoso para reconhecer tais gramáticas.
  - Eliminar a recursividade.

Bibliografia

Leituras:

- A. M. A. Price, S. S. Toscani. **Implementação de Linguagens de Programação**: Compiladores. 3ª ed. Porto Alegre: Sagra-Luzzatto. 2005. Cap 3, até Seção 3.2.2.
- A. V. Aho, R. Sethi, J. D. Ullman. **Compilers**: Principles, Techniques and Tools. Reading: Addison-Wesley. 1985. Seção 4.4.