4- Análise Sintática Descendente

Linguagens e Programação

Ana Madureira

Engenharia Informática
Ano Letivo: 2021/2022

Fontes:

1. Compiladores Princípios e práticas, Keneth C.Louden, Thomson, 2004.
   Cap. 4 Análise sintáctica descendente
2. Compiladores - Princípios , Técnicas e Ferramentas, Alfred V. Aho, Monica S. Lam e Ravi Sethi, Pearson, 2ª edição, 2007.
   Cap. 4 secção 4.5 Análise Sintáctica Descendente
3. Processadores de Linguagens – da concepção à implementação, Rui Gustavo Crespo. IST Press.1998.
   Cap. 4 Análise sintáctica descendente

1

Classe de Analisadores Sintáticos

- Verificar se uma dada sequência de *tokens* constitui um programa válido
- **Descendentes (top-down)**
  - Com retrocesso
  - Sem retrocesso
    - Recursivo
    - Preditivo (LL)
- **Ascendentes (bottom-up)**
  - Shift-reduce
  - Shift-reduce com análise de precedência
  - LR
    - LR(0)
    - SLR(1)
    - LALR(1)
    - LR(1)

2

Análise *Top-Down*

- Neste tipo de análise sintática, a árvore de parse é construída partindo da raiz, até se chegar à sequência de *tokens* do texto da entrada:
  - Gera-se sempre uma derivação mais à esquerda
  - Os nós da árvore de parse construída são visitados em pré-ordem (descida recursiva) ou próximo;
- São usados essencialmente dois métodos para implementar a análise *top-down*:
  - O método da descida recursiva, em que se associa uma rotina a cada não-terminal da gramática que é chamada quando se pretende reescrever esse símbolo;
  - O método da análise preditiva não-recursiva, que necessita de um stack auxiliar e é guiada por uma tabela de parse.
- Estes tipos de análise sintáctica requerem geralmente uma classe de gramáticas livres de contexto, denominadas LL(k) (na prática LL(1))
  - O método da descida recursiva, sendo geralmente escrito "à mão", pode incorporar técnicas ad-hoc e portanto relaxar um pouco as exigências das gramáticas LL(k).

3

Análise T*op-Down - Exemplo*

Considere-se a gramática:

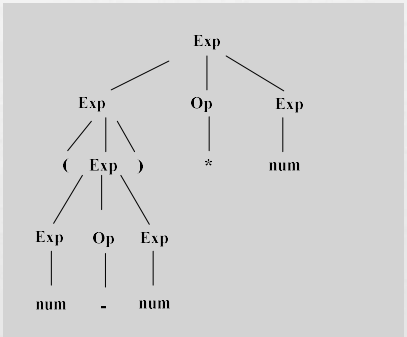
Exp → Exp Op Exp | (Exp) | num
Op → + | - | * | /

E a sequência de tokens:

(num - num) * num

Numa análise **top-down** partimos do símbolo inicial (Exp) e tentamos chegar à sequência de tokens através de uma derivação geralmente mais à esquerda:

Exp ⇒ Exp Op Exp
⇒ (Exp) Op Exp
⇒ (Exp Op Exp) Op Exp
⇒ (num Op Exp) Op Exp
⇒ (num - Exp) Op Exp
⇒ (num - num) Op Exp
⇒ (num - num) * Exp
⇒ (num - num) * num



4

Analisadores Sintáticos Descendentes

- Constrói a derivação mais à esquerda da sentença de entrada a partir do símbolo inicial da gramática.
- Podem ser implementados:
  - com retrocesso (analisadores não-determinísticos)
    - Força bruta
    - Maior conjunto de gramáticas (ambíguas)
    - Maior tempo para análise
    - Dificuldade para recuperação de erros
    - Dificuldade para análise semântica e geração de código
  - sem retrocesso (analisadores determinísticos)
    - Classe limitada de gramáticas (LL1)

5

Analisadores Sintácticos Descendentes - Dificuldades

- Quando mais do que uma regra, para o mesmo símbolo não-terminal, começam pelo mesmo símbolo é difícil escolher uma das regras, examinando apenas o próximo *token*
- **Recursividade à esquerda: (A → + Aα)**
  - Pode levar a ciclos infinitos!
  - Solução: modificar gramática de modo a **eliminar recursividade à esquerda.**
- **Produções que começam com o mesmo símbolo: (A → Xα | Xβ)**

  dificuldade em decidir qual das produções aplicar

  Solução: Atrasar a decisão – **Factorização à esquerda**

6

Eliminação da recursividade à esquerda

- Uma gramática é recursiva à esquerda se possuir um não terminal A, tal que exista uma derivação $A \Rightarrow^{+} A\alpha$, $A \in V$ e $\alpha \in \Sigma$, para alguma cadeia α.
- Suponha uma gramática com regras de produção na forma:

  $A \rightarrow A\alpha$ onde $A \in V$ e β não começa por A
  $A \rightarrow \beta$
- Esta gramática é recursiva à esquerda. Para ser eliminada aplica-se o seguinte procedimento:
  - **Agrupam-se todas as produções A, na forma**

    $A \rightarrow A\alpha \mid \beta$ onde β não começa por A
  - **Substitui-se, de forma equivalente, o conjunto de regras, pelo seguinte conjunto:**

    $A \rightarrow \beta A'$
    $A' \rightarrow \alpha A' \mid \varepsilon$

Exemplo: Sejam as regras para expressão do exemplo anterior:
E → E + T
E → E - T
E → T
Podem ser transformadas no conjunto:
E → T E' **E' → + T E'**
E' → - T E'
E' →ε

7

Eliminação da recursividade à esquerda geral

- Existem formas de recursividade à esquerda não imediatas
  - Existe recursividade à esquerda quando $A \rightarrow^{+} A\alpha$
  - Exemplo:

    $S \rightarrow Aa \mid b$ $A \rightarrow Ac \mid Sd \mid \varepsilon$ $S \rightarrow Aa \mid Sda$
- Estas formas não ocorrem geralmente nas gramáticas das linguagens de programação

1. Ordenar os não-terminais numa ordem arbitrária: $A_1, A_2, \ldots, A_i, \ldots, A_n$
2. for i de 1 a n do
   for j de 1 a i-1 do
   substituir todas as produções da forma $A_i \rightarrow A_j\gamma$ pelas produções
   $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, onde
   $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ são todas as produções decorrentes de A_j
   eliminar, se for caso disso, a recursividade à esquerda imediata de A_i

8

Factorização à esquerda

- Quando mais do que uma regra, para o mesmo não-terminal, começam pelo mesmo símbolo é difícil escolher uma das regras, examinando apenas o próximo *token*
- Solução: atrasar a escolha até haver distinção
- Algoritmo de factorização à esquerda:

1. Para cada não-terminal A da gramática encontrar o prefixo mais longo α, comum a duas ou mais das suas regras
2. Se $\alpha \neq \varepsilon$, substituir todas as produções de $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, onde γ representa regras que não começam por α por:
   $A \rightarrow \alpha A' \mid \gamma$
   $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
3. Repetir o processo até $\alpha = \varepsilon$, para todas as regras, ou seja, até que todas as regras no mesmo não-terminal comecem com símbolos diferentes

Exemplo:
Seja a gramática: S → if E then S | if E then S else S
Aplicando a factorização à esquerda obtemos:
S → if E then S X
X → else S| ε

9

Descida recursiva

- Consiste na construção de um conjunto de procedimentos (normalmente recursivos), um para cada símbolo não terminal da gramática em questão.
- A cada não-terminal está associada uma rotina, sendo a sua análise efectuada pela chamada a essa rotina.
- Considere-se a seguinte gramática para uma linguagem muito simples:

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

- A esta gramática deve ser acrescentado um novo símbolo inicial P e a seguinte regra:
  P → S $ ($ designa o fim do texto – EOF)
- Será necessário adicionar uma rotina relativa a esta nova produção

10

Descida recursiva

```
enum token {EOF, IF, THEN, ELSE, BEGIN,
END,PRINT, SEMI, NUM, EQ};
enum token getToken(void);
enum token tok;
void advance(void) { tok = getToken(); }
void eat(enum token t)
{ if (tok == t) advance();
  else error();
}
void P(void) {
  S();
  if (tok == EOF) accept();
  else error();
}
void S(void) { switch (tok) {
  case IF:
     eat(IF); E(); eat(THEN); S(); eat(ELSE);
     S();
     break;
  case BEGIN:
     eat(BEGIN); S(); L(); break;
  case PRINT:
     eat(PRINT); E(); break;
  default: error();
} }
```

P → S $
S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

```
void L(void) { switch(tok) {
  case END:
     eat(END); break;
  case SEMI:
     eat(SEMI); S(); L(); break;
  default: error();
} }

void E(void) { eat(NUM); eat(EQ); eat(NUM);
}

void main(void)
{
  advance(); P();
}
```

11

Descida recursiva

- **Desvantagens**
  - não é geral, ou seja, os procedimentos são específicos para cada gramática
  - tempo de análise é maior
  - necessidade de uma linguagem que permita recursividade para sua implementação
  - Dificuldade de validação
- **Vantagens**
  - simplicidade de implementação
  - facilidade para inserir as diferentes funções do processo de compilação nomeadamente a possibilidade de inclusão de rotinas de análise semântica – se existem diferentes alternativas para reescrever o mesmo não terminal.
  - eficiência – não será necessário tentar diferentes rotinas para o reconhecimento de uma frase (backtraking)

12

Descida recursiva - dificuldades

Consideremos agora a seguinte gramática para expressões aritméticas:

S → E $ **T → T * F** **F → id**
E → E + T **T → T / F** **F → num**
E → E - T **T → F** **F → (E)**
E → T

Como escrever as rotinas relativas a E e T ?
Não sabemos como distinguir as várias produções. Mesmo sabendo, as primeiras 2 produções de E e T produziriam chamadas recursivas infinitas.
Se escrevermos a gramática anterior na forma EBNF obtemos:

S → E $ **T → F { *|/ F }** **F → id**
E → T { +|- T } **F → num** **F → (E)**

As expansões para E e T tornam-se evidentes:

```
void E(void) {
   T(); while (tok==PLUS || tok==MINUS) {
      advance(); T(); }
}
void T(void) {
   F(); while (tok==MUL || tok==DIV) {
      advance(); F(); }
}
```

No entanto existem métodos para eliminar a recursividade à esquerda de uma gramática e para distinguir entre regras, quando um não-terminal possui várias produções que começam por não-
-terminais distintos, ou símbolos iguais.
São eles:

Recursividade à esquerda:
(A ⁺ Aα)
Fazer uma eliminação da recursividade à esquerda, transformando a gramática

Produções que começam com o mesmo símbolo:
(A → Xα | Xβ)
Fazer uma factorização à esquerda

Produções que começam por não-terminais distintos:
(A →Bα | Cβ)
Dispor do conjunto de terminais que pode começar qualquer derivação de B e C - starters(B) e starters(C)

Produções vazias:
(A →ε)
Dispor do conjunto de terminais que pode legalmente aparecer a seguir a A em qualquer forma sentencial - followers(A)

13

Informação sobre gramáticas

- É necessário obter informação sobre símbolos não terminais em GIC, para decidir quais as produções a serem usadas durante o processo de análise sintáctica:
  - Se o símbolo não terminal A gera ou não a cadeia vazia ε
  - Quais são os símbolos terminais iniciadores (Starters(A)) das frases geradas a partir de A:

    Se $A \Rightarrow^* \alpha$, que terminais podem aparecer como primeiro símbolo de α
  - Quais são os símbolos terminais seguidores (Followers(A)): ou seja, se $A \Rightarrow^* \alpha A \beta$, que terminais podem aparecer como primeiro símbolo de β

14

Conjuntos Starters e Followers

- **Definições:**

Para o não-terminal X define-se:

- **nullable(X)** - função que retorna TRUE se $X \Rightarrow^+ \varepsilon$ e FALSE no caso contrário
- **Starters(X)** - conjunto de terminais que iniciam as frases deriváveis de X -

  Starters(X) = { t | $X \Rightarrow^* t\alpha$ }
- **Followers(X)** - conjunto de terminais que podem aparecer a seguir a um X em qualquer forma sentencial - **Followers(X) = { t | $S \Rightarrow^* \alpha X t \beta$ }**

- Seguindo as definições de cima, são válidas as afirmações:

  Para uma regra $A \rightarrow X1\ X2 \ldots Xn$ e para $1 \le i < j \le n$
  Starters(A) ⊇ Starters(X1)
  Followers(Xn) ⊇ followers(A)
  Followers(Xi) ⊇ Starters(Xi+1)
  se nullable(Xk) para $1 \le k \le n$ então nullable(A) = TRUE
  se nullable(Xk) para qualquer $1 \le k \le i-1$ então Starters(A) ⊇ Starters(Xi)
  Admite-se que Starters(t) = {t}, t – terminal
  se nullable(Xk) para qualquer $i+1 \le k \le n$ então followers(Xi)⊇followers(A)
  se nullable(Xk) para qualquer $i+1 \le k \le j-1$ então Followers(Xi)⊇Starters(Xj)

15

Utilização dos *starters* e dos *followers* para implementação de um analisador de descida recursiva (exemplo)

Verificou-se anteriormente que a gramática para expressões não era adequada para a descida recursiva (era recursiva à esquerda).

Eliminando a recursividade à esquerda daquela gramática, obtém-se:

Exemplo: Sejam as regras para expressão do exemplo anterior:

E → E + T
E → E - T
E → T

Podem ser transformadas no conjunto:

E → T E'
E' → + T E'
E' → - T E'
E' →ε

S →E $
E→T E'
E' →+ T E'
E' →- T E'
E' →ε
T →F T'
T' →* F T'
T' →/ F T'
T' →ε
F →id
F →num
F →(E)

que já seria adequada para implementar uma descida recursiva.

Assim as rotinas de descida recursiva correspondentes a E' e T' seriam:

```
void Eprime(void) {
   switch (tok) {
     case PLUS:
       eat(PLUS); T(); Eprime(); break;
     case MINUS:
       eat(MINUS); T(); Eprime(); break;
     case RPAR: case EOF: break;
     default: error();
   }
}
void Tprime(void) {
   switch (tok) {
     case MUL:
       eat(MUL); F(); Tprime(); break;
     case DIV:
       eat(DIV); F(); Tprime(); break;
     case RPAR: case PLUS: case MINUS: case
EOF:
       break;
   default: error();
   }
}
```

16

Utilização dos *starters* e dos *followers* para implementação de um analisador de descida recursiva (exemplo)

Verificou-se anteriormente que a gramática para expressões não era adequada para a descida recursiva (era recursiva à esquerda).

Eliminando a recursividade à esquerda daquela gramática, obtém-se:

S →E $		
E→T E'	T →F T'	
E' →+ T E'	T' →* F T'	F →id
E' →- T E'	T' →/ F T'	F →num
E' →ε	T' →ε	F →(E)

que já seria adequada para implementar uma descida recursiva. No entanto se pretendermos validar as produções vazias de E' e T', necessitamos dos seus followers:

followers(E') = {), $ }
followers(T') = {), +, -, $ }

Assim as rotinas de descida recursiva correspondentes a E' e T' seriam:

```
void Eprime(void) {
   switch (tok) {
     case PLUS:
       eat(PLUS); T(); Eprime(); break;
     case MINUS:
       eat(MINUS); T(); Eprime(); break;
     case RPAR: case EOF: break;
     default: error();
   }
}
void Tprime(void) {
   switch (tok) {
     case MUL:
       eat(MUL); F(); Tprime(); break;
     case DIV:
       eat(DIV); F(); Tprime(); break;
     case RPAR: case PLUS: case MINUS: case
EOF:
       break;
   default: error();
   }
}
```

17

Construção explícita da árvore de parse em descida recursiva

- Para a construção da árvore de parse numa decida recursiva; basta que cada rotina retorne um apontador para a subárvore que representa.
- Admitindo que dispomos da função makenode() que aloca um nó de árvore, preenche a respectiva informação e retorna o apontador para esse nó, as rotinas para E' e F da gramática anterior ficariam:

```
Typedef struct node {
  enum kind info;
  struct node *child[MAX_CHILD];
} *tree;
```

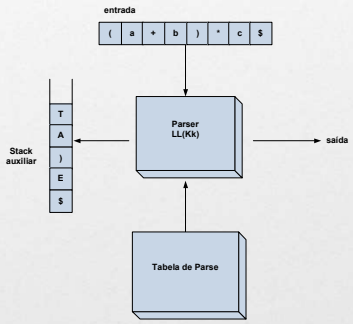
```
tree Eprime(void) {
  tree t = makenode(EXP_P);
  if (tok == PLUS)
   {
    t->child[0] = makenode(PLUS); advance();
    t->child[1] = T();
    t->child[2] = Eprime();
    } elseif (tok == MINUS)
     {
       t->child[0] = makenode(MINUS); advance();
       t->child[1] = T();
       t->child[2] = Eprime();
     } else   t->child[0] = makenode(EPSILON);
   return t;
}
```

```
tree F(void) {
tree t = makenode(FACTOR);
switch(tok) {
case ID:
        t->child[0] = makenode(ID); advance(); break;
case NUM:
        t->child[0] = makenode(NUM); advance(); break;
case LPAR:
        t->child[0] = makenode(LPAR); advance();
        t->child[1] = E(); eat(RPAR);
        t->child[2] = makenode(RPAR); break;
default:  error();
}
return t;
}
```

18

Analisadores sintácticos preditivos (não recursivos)

- Estes analisadores sintácticos utilizam uma tabela, que dirige as regras a aplicar, além de um ***stack*** auxiliar de símbolos gramaticais e a consulta de um ou mais dos ***tokens*** da entrada



LL(k):
L - left to right parse
L - leftmost derivation
k - número de *tokens* de *lookahead*
(na prática k = 1)

19

Tabela de Parse

- Em cada entrada da tabela M existe uma única produção viabilizando a análise determinística da sentença de entrada.
- Para isso é necessário que a gramática:
  - Não possua recursividade à esquerda
  - Estar fatorizada (é determinística)
  - Para todo $A \in \sum \mid A \Rightarrow^* \varepsilon$ e, $First(A) \cap Follow(A) = \phi$

20

Tabela de parsing

- As GLC que satisfazem estas condições são denominadas GLC LL(K)
  - podem ser analisadas deterministicamente da esquerda para a direita (Left-to-right)
  - o analisador construirá uma derivação mais à esquerda (Leftmost derivation)
  - sendo necessário a cada passo o conhecimento de K símbolos de lookahead (símbolos de entrada que podem ser vistos para que uma ação seja determinada).
- Somente GLC LL(K) podem ser analisadas pelos analisadores preditivos (os analisadores preditivos são também denominados analisadores LL(K)
  - na prática usa-se K = 1, obtendo-se desta forma Analisadores LL(1) para GLC LL(1).

21

Analisadores sintáticos preditivos (não recursivos)

L L(1) – analisadores preditivos

- exactamente 1 símbolo da entrada é examinado (*lookahead*)
- o analisador constrói uma derivação esquerda ***Leftmost***
- a cadeia de entrada é examinada da esquerda para a direita *Left-to-right*

- Objetivo do mecanismo LL(1) – selecção da regra a utilizar durante o processo de análise descendente, baseada em:
  - próximo símbolo não-terminal A a ser expandido
  - primeiro símbolo 'a' do resto da entrada a analisar

22

Funcionamento de um parser preditivo

- **Configuração inicial:**
  - Stack auxiliar: contém no fundo o símbolo $ (eof) e no topo o símbolo S
  - **Entrada**: nada ainda consumido, o primeiro token está acessível
- **Tabela de parse**
  - É uma matriz da forma M[N, T] onde N (linhas) é o conjunto dos não-terminais e T (colunas) é o conjunto dos terminais mais o símbolo $
  - Cada entrada da tabela contém uma regra gramatical ou a indicação de erro
- **Acções do parser:**
  - Seja X o símbolo que está no topo do stack e seja z o próximo *token* disponível;
  - Se X = z = $ então o texto da entrada é aceite (conclui a análise sintáctica)
  - Se X = z ≠ $ então:
    - fazer o pop() da stack (descartar X)
    - avançar a entrada (descartar z)
  - Se X = não-terminal
    - consultar M[X, z] - se M[X, z] contiver uma regra:

      fazer pop() (retirar X da stack)

      fazer push do lado direito da produção que estiver na tabela por ordem inversa

      (Se na tabela estiver $X \rightarrow \alpha\beta\gamma$, fazer push($\gamma$); push($\beta$); push($\alpha$);)

      indicar a aplicação dessa regra (p. ex. printf("$X \rightarrow \alpha\beta\gamma$");)
  - Outras situações: erro

23

Algoritmo parser preditivo

```
push($);
push(S);
advance();
while (top() != $  &&  tok != $) do
     if  ((a = top()) == terminal  &&  tok == a)  then
            pop();
            advance();
     else
        if ((X = top()) == non-terminal  &&  M[X, tok] == X → Y1 ... Yn) then
            pop();
            for (k = n; k > 0; k--)      push(Yk);
            output(" X → Y1 ... Yn");
        else
            error();
if (top() == $  &&  tok == $) then
         accept();
else
         error();
```

24

Exemplo

Considere-se a gramática

E→T E' **T →F T'**
E' →+ TE' **T' →* FT'** **F →id**
E' →ε **T' →ε** **F →(E)**

Esta gramática, dá origem à seguinte tabela de parse:

	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → id			F → (E)		

Análise da entrada id + id * id $ e regras aplicadas para a aceitação desta

Stack	Entrada	Saída
$E	id + id * id $	
$E'T	id + id * id $	E → T E'
$E'T'F	id + id * id $	T → F T'
$E'T'id	id + id * id $	F → id
$E'T'	+ id * id $	
$E'	+ id * id $	T' → ε
$E'T+	+ id * id $	E' → + T E'
$E'T	id * id $	
$E'T'F	id * id $	T → F T'
$E'T'id	id * id $	F → id
$E'T'	* id $	
$E'T'F*	* id $	T' → * F T'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	T' → ε
$E'	$	E' → ε
$	$	

25

Exemplo

Considere-se a seguinte gramática para instruções:

S →If-st | other
If-st →if (Exp) S El
El →else S | ε
Exp →0 | 1

	Nullable	First	Follow
S	false	if, other	else, $
If-st	false	if	else, $
Else-part	true	else	else, $
Exp	false	0, 1	)

- **Contruir a tabela de parse para esta gramática.**
- **Começando por determinar nullable, starters e followers chegaríamos aos seguintes resultados:**

	if	other	else	0	1	$
S	S → I	S → o				
If-st	I → if(E)S El					
El			El → e S El → ε			El → ε
Exp				E → 0	E → 1	

- Repare-se na dupla entrada em M[El, else] que faz com que esta gramática não seja LL(1).
- Deve-se à conhecida ambiguidade da instrução if com else opcional.
- No entanto usando a habitual regra de elimição de ambiguidade (emparelhar cada else com o último if) a produção El →ε pode ser eliminada dessa entrada, tornando a tabela perfeitamente definida e correcta para aquela regra.

26

Análise sintática LL(1)
Construção da tabela M[A, a]

- Para todas as produções A → α da gramática:
  - Para cada regra i: A → α, temos M[A, a] = i, para cada 'a' em Starters(α)
  - Para cada regra i: A → α, se α →* ε, temos M[A, a] = i, para cada 'a' em Followers(A)
- As entradas de M[A, a] que não recebem qualquer valor devem ser marcadas como entradas de erro (–)

M[A, a]	(	a	+	*	)	$
E	1	1	–	–	–	–
T	2	2	–	–	–	–
F	3	4	–	–	–	–
E'	–	–	5	–	6	6
T'	–	–	8	7	8	8

1. E → TE'	**S(TE') = { (, a }**	**M[E, ( ] = M[E, a] = 1**
2. T → FT'	**S(FT') = { (, a }**	**M[T, ( ] = M[T, a] = 2**
3. F → (E)	**S((E)) = { (}**	**M[F, ( ] = 3**
4. F → a	**S(a) = { a }**	**M[F, a] = 4**
5. E' → +TE'	**S(+TE') = { + }**	**M[E', +] = 5**
6. E' → ε	**F(E') = { $,) }**	**M[E', $] = M[E',)] = 6**
7. T' → +FT'	**S(*FT') = { * }**	**M[T', *] = 7**
8. T' → ε	**F(T') = { $, +,) }**	**M[T', $]=M[T', +]=M[T',)] = 8**

Condição para a gramática ser LL(1):
Para cada entrada M[A, a] só pode haver um valor; caso contrário, diz-se que existe um conflito e a gramática não é LL(1)

27

Transformação de uma gramática não LL(1) numa gramática LL(1) (1/3)

- Em alguns casos conclui-se que uma gramática não é LL(1). As duas características mais óbvias são a:
  - **recursividade à esquerda** – uma gramática é recursiva à esquerda se permite uma derivação do tipo:
    $A \rightarrow^* A\alpha$, $A \in V_{NT}$, $\alpha \neq \varepsilon$
    $A \rightarrow A\alpha$
    $A \rightarrow \beta$ (Para que A não seja inútil é necessário pelo menos uma regra $A \rightarrow \beta$ sem recursividade à esquerda)
    Esta gramática gera cadeias da forma βαααα… Mas estas cadeias podem ser geradas de forma alternativa (eliminando a recursividade à esquerda):
    $A \rightarrow \beta A'$
    $A' \rightarrow \alpha A'$
    $A' \rightarrow \varepsilon$ (Apresenta recursividade à direita, que não cria problemas)
  - **possibilidade de factorização**
    - Se $\alpha \neq \varepsilon$, substituir todas as produções de $A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \ldots \mid \alpha\beta n \mid \gamma$, onde γ representa regras que não começam por α por:
      $A \rightarrow \alpha A' \mid \gamma$
      $A' \rightarrow \beta1 \mid \beta2 \mid \ldots \mid \beta n$
    - **Exemplo: $A \rightarrow aA \mid aB$**
      $A \rightarrow aA'$
      $A' \rightarrow A \mid B$

28

Transformação de uma gramática não LL(1) numa gramática LL(1) (2/3)

- **Possibilidade de factorização** – Considere-se a situação em que duas regras começam pelos mesmos símbolos, isto é, regras como A → αβ e A → αγ, com S(α) ≠ ∅.

  Neste caso, existe uma intersecção entre S(αβ) e S(αγ), e a gramática não pode ser LL(1). Isto acontece porque não é possível decidir, olhando apenas para o primeiro símbolo derivado de α, qual a regra correcta. A solução envolve a factorização: a cadeia inicial α é *posta em evidência*.
  Temos: A → αA'

  A' → β | γ

  Isto implica adiar a decisão entre β e γ para quando o primeiro símbolo derivado de β ou de γ estiver visível.

29

Transformação de uma gramática não LL(1) numa gramática LL(1) (3/3)

Estas duas técnicas (**eliminação de recursividade à esquerda, factorização**) permitem transformar algumas gramáticas em gramáticas LL(1). No entanto, algumas gramáticas independentes de contexto não têm gramáticas equivalentes LL(1), e nesse caso a aplicação destas (ou de outras) técnicas não poderá ter sucesso.

Exemplo: Transformação da gramática seguinte numa gramática LL(1) equivalente

L → L ; S | S

S → if E th L el L fi

| if E th L fi | s

E → e

Frase: if e th s fi
L → S → if E th L fi → if e th S fi → if e th s fi

Existe recursividade à esquerda nas regras de L e possibilidade de factorização nas regras de S. O resultado da transformação é a gramática

L → SL'

L' → ; S L' | ε

S → if E th L S' | s

S' → el L fi | fi

E → e

Frase: if e th s fi
L → SL' → if E th LS'L' → if e th LS'L' →
→ if e th SL'S'L' → if e th s L'S'L' → if e th s S'L' →
→ if e th s fi L' → if e th s fi

É possível verificar que é uma gramática LL(1).

30

Tratamento de erros

- O tratamento de erros durante a análise sintáctica deve obedecer a certos princípios gerais:
  - O erro deve ser declarado o mais cedo possível, de modo a que possa ser localizado com precisão na sequência de tokens da entrada
  - Depois de detectado um erro o parser deve continuar, num ponto mais à frente, de modo a detectar o maior número possível de erros numa única passagem
  - O parser deve tentar evitar o problema da "cascata de erros", em que um único erro gera uma série de mensagens
  - O parser deve evitar um ciclo infinito, reatando a análise sem consumir nenhum token, e gerando sucessivamente as mesmas mensagens de erros
- A forma usual de tratamento de erros em *parsers top-down* é tentar sincronizar a análise a partir de um próximo *token*, descartando alguns se for necessário, quando se detecta um erro sintáctico
- Estes *tokens* de sincronização, deverão ser criteriosamente escolhidos para cada classe gramatical e são geralmente *tokens* dos conjuntos *followersers( )* e *starters( )* das classes gramaticais mais significativas da linguagem

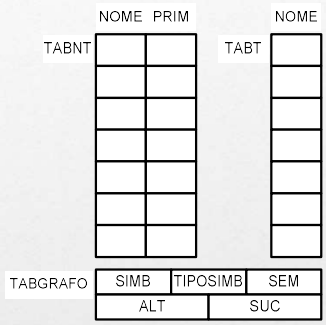
31

Tratamento de erros (parsers preditivos)

- Nos parsers preditivos detecta-se um erro quando temos um não-terminal (A) no topo da *stack* e um *token* de entrada que não pertence a Starters(A) ou followers(A) se nullable(A)
  - A situação de aparecer um terminal no topo da stack diferente da entrada nunca surge
- Nas situações de erro, e para as recuperar podemos executar uma de 3 acções:
  - Retirar A da *stack* (pop)
  - Esta alternativa é escolhida se o próximo token for $ ou se pertencer a followers(A)
    - Descartar sucessivamente *tokens* da entrada (scan), até se poder continuar
  - Escolhe-se esta alternativa nos casos não contemplados na alternativa anterior
    Vão-se descartando tokens da entrada até se encontrar um pertencente a Starters(A) ∪ followers(A)
    - Colocar (push) um não-terminal na *stack*
  - Se em virtude da primeira acção a stack ficar vazia, normalmente faz-se o push do símbolo inicial da gramática
- Estas acções são codificadas na tabela de parse

32

Construção de um analisador Analisador baseado em tabelas e grafos sintáticos (1/3)



TABNT: tabela de símbolos não-terminais
TABT: tabela de símbolos terminais
TABGRAFO: tabela de ligações entre símbolos
NOME: identificação de símbolo (terminal ou não-terminal)
PRIM: apontador para o símbolo que inícia o lado direito de uma regra
SIMB: apontador para célula de TABNT ou TABT
TIPOSIMB: tipo de símbolo – terminal ou não-terminal
SEM: rotina semântica a executar após o reconhecimento sintáctico
ALT: apontador para regra alternativa
SUC: apontador para o próximo símbolo

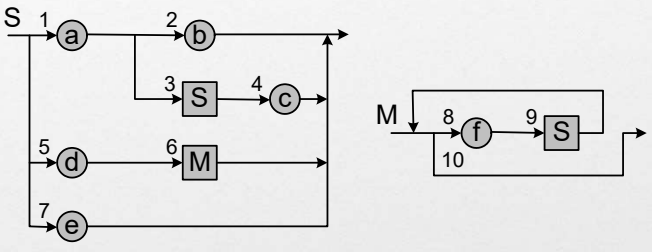
- Cada regra é representada por uma lista ligada em que os elementos são do tipo TABGRAFO
- As listas ligadas referentes a regras alternativas do mesmo não-terminal estão interligadas através do apontador ALT

33

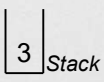
Construção de um analisador Analisador baseado em tabelas e grafos sintácticos (2/3)

S → ab | aSc | dM | e
M → { fS }*

Grafo sintáctico:

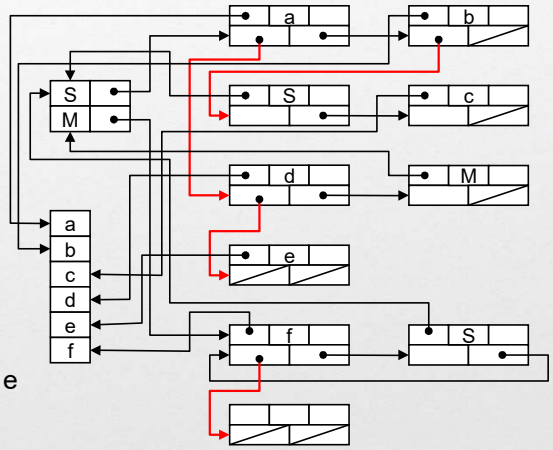


Quando se efectuam chamadas a outros símbolos não-terminais, é necessário usar uma "stack" para suporte da análise: depois da chamada é necessário voltar ao ponto de onde foi efectuada a chamada.
Exemplo: Frase aabc S → aSc → a ab c



34

Construção de um analisador Analisador baseado em tabelas e grafos sintácticos (3/3)



S → ab | aSc | dM | e
M → { fS }*

35