CTC-41

Compiladores

Carlos Alberto Alonso Sanches

CTC-41

4) Análise sintática

Análise sintática descendente (*top-down*)

Parsers preditivos recursivos e LL(1)

Análise sintática descendente

- Na análise sintática descendente (*top-down*):
  - A árvore de derivação é construída a partir da raiz (que é o símbolo inicial da gramática), chegando até as folhas (que são os *tokens*).
  - Em cada passo desta estratégia, um lado esquerdo de uma produção gramatical (um não-terminal) é substituído por um lado direito.
  - Por isso, o reconhecimento segue uma derivação mais à esquerda.
- Principais *parsers* descendentes:
  1) Com retrocesso (*backtracking*): testam diferentes possibilidades de análise da entrada, retrocedendo quando houver falha. Podem ser recursivos ou não. Como tendem a ser mais lentos, não serão estudados.
  2) Preditivos: tentam prever a análise utilizando um ou mais *tokens* à frente. Estudaremos os dois casos abaixo:
     - *Parsers* recursivos: são mais fáceis de implementar, utilizados para linguagens simples e suficientemente formalizadas.
     - *Parsers* não-recursivos (ou tabulares): são mais complexos, pois utilizam uma tabela preditiva e uma pilha.

Parsers preditivos recursivos

- Em um *parser* preditivo recursivo, cada símbolo não-terminal da gramática livre de contexto possui um procedimento recursivo que verifica se os *tokens* de entrada correspondem ou não às suas produções gramaticais.
- A codificação de cada procedimento recursivo é feita de acordo com o lado direito das produções gramaticais:
  - Os terminais devem "casar" com os *tokens* de entrada;
  - Cada não-terminal será tratado com a chamada do correspondente procedimento recursivo;
  - As alternativas de cada produção gramatical são selecionadas através de comandos "if" ou "case".

Exemplo 1

- Considere a gramática abaixo:
  - T = {**BEGIN**, **IF**, **THEN**, **ELSE**, **PRINT**, **ID**, **END**, **NUM**, **;**, **=**}
  - **S → BEGIN S L | IF E THEN S ELSE S | PRINT ID**
    L → ; S L | END
    E → ID = NUM
- *Parser* preditivo recursivo para esta gramática:
  - **getToken()**: retorna o próximo *token* fornecido pelo analisador léxico
  - **Erro()**: imprime mensagem de erro

```
void main() {
  tok = getToken();
  S();
}

void S() {
  switch(tok) {
     case BEGIN: match(BEGIN); S(); L(); break;
     case IF:    match(IF); E(); match(THEN); S(); match(ELSE); S(); break;
     case PRINT: match(PRINT); match(ID); break;
     default:    Erro();
}

void E() {
  match(ID); match("="); match(NUM);
}
```

```
void match(int t) {
  if (tok == t) tok = getToken();
  else Erro();
}

void L() {
  switch(tok) {
     case ";":  match(";"); S(); L(); break;
     case END:  match(END); break;
     default:   Erro();
  }
}
```

Exemplo 2a

- Considere a gramática abaixo:
  - T = {`(`, `)`, `+`, `-`, `*`, `/`, **`NUM`**}
  - ```
    exp → exp op1 termo | termo
    op1 → + | -
    termo → termo op2 fator | fator
    op2 → * | /
    fator → ( exp ) | NUM
    ```
- Veja como ficaria o procedimento recursivo para **`exp`**:

```
void exp() {
  exp(); op1(); termo();
}
```

- Há dois problemas nesta codificação:
  - Infinitas chamadas recursivas de **`exp()`**: é o problema da *recursão à esquerda*
  - Não há como distinguir as duas alternativas da produção gramatical de **`exp`**
- Uma solução é reescrever esta gramática no formato *Extended BNF* (EBNF), que permite o uso de chaves e colchetes:
  - { ... }: zero ou mais vezes (corresponde ao * das ER)
  - [...]: opcional (corresponde ao ? das ER)
```

Exemplo 2b

- Considere a gramática anterior reescrita em formato EBNF:
  - T = {(,), +, -, *, /, NUM}
  - exp → termo {op1 termo}
    op1 → + | -
    termo → fator {op2 fator}
    op2 → * | /
    fator → (exp) | NUM
- *Parser* preditivo recursivo para esta gramática:

```
void exp() {
  termo();
  while ((tok == "+") || (tok == "-")) {
     match(tok); termo();
  }
}
```

```
void termo() {
  fator();
  while ((tok == "*") || (tok == "/")) {
     match(tok); fator();
  }
}
```

```
void fator() {
  switch(tok) {
     case "(": match("("); exp(); match(")"); break;
     case NUM: match(NUM); break;
     default:  Erro();
}
```

```
void main() {
  tok = getToken();
  exp();
}
```

Exemplo 2b: árvore sintática

- Considere a mesma gramática em formato EBNF:
  - T = {(,), +, -, *, /, NUM}
  - ```
    exp → termo {op1 termo}
    op1 → + | -
    termo → fator {op2 fator}
    op2 → * | /
    fator → ( exp ) | NUM
    ```
- Em um *parser* preditivo recursivo, é simples inserir código para a geração da árvore sintática.
- Exemplo para **exp**:

```
tree exp() {
     tree t1, t2;
     t1 = termo();
     while ((tok == "+") || (tok == "-")) {
          t2 = NodeOp(tok);
          match(tok);
          t2->left = t1;
          t2->right = termo();
          t1 = t2;
     }
     return t1;
}
```
```

Exemplo 3

- Considere a produção gramatical abaixo:
  - `if-stat → IF (exp) stat | IF (exp) stat ELSE stat`
- Problema: há duas alternativas que começam com o *token* `IF`
- Mesma produção reescrita em formato EBNF:
  - `if-stat → IF (exp) stat [ ELSE stat ]`
- *Parser* preditivo recursivo para esta produção:

```
void if-stat() {
  match(IF); match("("); exp(); match(")"); stat();
  if (tok == ELSE) {
     match(tok); stat();
  }
}
```

É semelhante à eliminação de ambiguidade com aninhamento mais próximo

- Com geração de árvore sintática:

```
tree if-stat() {
     match(IF); match("(");
     tree t1 = NodeStat(IF);
     t1->test = exp(); match(")"); t1->then = stat();
     if (tok == ELSE) {
          match(tok); t1->else = stat();
     }
     else t1->else = NULL;
     return t1;
}
```

Parsers preditivos não-recursivos

- Como vimos nos exemplos anteriores, a construção de um *parser* recursivo exige que todas as produções gramaticais tenham um primeiro símbolo terminal que permita uma escolha segura.
- Contraexemplos:
  - $A \rightarrow \alpha \mid \beta$, onde α e β começam com símbolos não-terminais
  - $A \rightarrow \varepsilon$: como esse não-terminal pode desaparecer, é preciso saber quais *tokens* podem eventualmente sucedê-lo.
- Uma gramática livre de contexto é chamada de LL(1) quando pode ser analisada por um *parser* descendente preditivo, onde:
  - a sentença de *tokens* é fornecida da esquerda para a direita (**L***eft to right*);
  - é adotada a derivação mais à esquerda (**L***eftmost derivation*);
  - é considerado apenas um *token* à frente (*lookahead* = **1**).
- A sua implementação utiliza uma tabela preditiva e uma pilha, semelhante a um autômato de pilha. Estes *parsers* costumam ter melhor desempenho, pois evitam o *overhead* das recursões.

Exemplo

- A gramática abaixo gera cadeias de parênteses balanceados:
  - S → (S) S | ε
- Como exemplo, vamos analisar a cadeia ( )
  - Utilizaremos uma pilha explícita, que começa com os símbolos $ S
    - $ é um símbolo especial que indica final de entrada
  - Em cada passo da análise sintática, há duas possíveis ações com o topo da pilha:
    1) Se for não-terminal: é desempilhado, e uma das suas produções gramaticais é empilhada
    2) Se for terminal: é desempilhado se "casar" com o próximo *token*, que é consumido da entrada
  - A cadeia será aceita se toda a entrada for consumida e a pilha terminar com $

Passos	Pilha	Entrada	Ações
1	$ S	( ) $	S → (S) S
2	$ S) S (	( ) $	*token* (
3	$ S) S	) $	S → ε
4	$ S)	) $	*token*)
5	$ S	$	S → ε
6	$	$	aceitação

Tabela preditiva

- Na implementação de um *parser* LL(1), é criada uma tabela preditiva M[N,T], onde N é o conjunto de não-terminais e T é o conjunto dos terminais da gramática a ser reconhecida, cujo símbolo inicial é **S**.
- Dados **A** ∈ N e **a** ∈ T, M[**A**,**a**] indica a produção gramatical de **A** que será utilizada no seu reconhecimento quando **a** for o próximo *token* de entrada.
- Regras para o preenchimento de M[N,T]:
  - Se houver produção **A** → α e existir derivação $\alpha \Rightarrow^* \mathbf{a}\beta$, então adicione **A** → α a M[**A**,**a**]
    - Se **a** está na entrada, selecionaremos a produção que gere **a**
  - Se houver produção **A** → α e existirem derivações $\alpha \Rightarrow^* \varepsilon$ e **S** $\Rightarrow^* \beta$**Aa**γ, então adicione **A** → α a M[**A**,**a**]
    - Se **A** deriva ε e **a** sucede **A** em alguma derivação, selecionaremos a produção que faça **A** desaparecer

Gramática LL(1)

- Uma gramática livre de contexto será LL(1) se a sua tabela preditiva tiver no máximo uma produção gramatical em cada posição.
- Exemplo:
  - T = {a, b, c}
  - S → a S A b | b A a
    A → b A b | c
- Tabela preditiva para esta gramática:

M	a	b	c
S	S → aSAb	S → bAa	
A		A → bAb	A → c

- Suas eventuais posições vazias indicam erros sintáticos, ou seja, os terminais que não podem aparecer no início de cada produção gramatical.
- *Lookahead* = 1 significa que apenas um terminal é suficiente para prever a próxima produção gramatical. Há gramáticas que exigem mais...

Pseudocódigo de um *parser* LL(1)

```
tok = getToken;                          // próximo token da entrada
stack p;
p.push(S);                               // empilha símbolo inicial da gramática
while (!p.isEmpty() && tok != ϕ)                 // ϕ significa entrada vazia
        if (p.top == tok) then {                 // "casou" com terminal
                p.pop();
                tok = getToken;
        }
        else if (p.top == A && M[A,tok] == A → X1X2...Xn) then {
                        p.pop();      // troca não-terminal por produção
                        for (i=n; i>0; i--)
                                p.push(Xi);
            }
            else Erro();
if (p.isEmpty() && tok == ϕ) then "Entrada aceita";
else Erro();
```

- Para este *parser* funcionar, a gramática precisa:
  - Não ter recursão à esquerda: caso contrário, o *parser* poderia entrar em *loop* infinito...
  - Ter uma tabela preditiva com uma única produção gramatical em cada posição.

Remoção de recursividade à esquerda

- A recursividade à esquerda é imediata quando houver alguma produção com formato $B \rightarrow B\alpha$, onde $B \in N$ e $\alpha \in (T \cup N)^+$.
- Também pode ser não-imediata: $B \rightarrow A\alpha$, $A \rightarrow B\beta$, onde $A,B \in N$ e $\alpha,\beta \in (T \cup N)^+$.
- Procedimento para remover recursividade imediata à esquerda de um não-terminal B:
  - Dividir as produções de B em dois subconjuntos:
    - Sem recursão: $\{\alpha_1, \alpha_2, ..., \alpha_n\}$
    - Com recursão: $\{B\beta_1, B\beta_2, ..., B\beta_m\}$
  - Eliminar da gramática as produções do subconjunto com recursão
  - Criar um novo não-terminal B', e adicionar à gramática as seguintes produções: $B \rightarrow \alpha_1 B'$, $B \rightarrow \alpha_2 B'$, ..., $B \rightarrow \alpha_n B'$.
  - Adicionar à gramática as seguintes produções: $B' \rightarrow \beta_1$, $B' \rightarrow \beta_2$, ..., $B' \rightarrow \beta_m$, $B' \rightarrow \beta_1 B'$, $B' \rightarrow \beta_2 B'$, ..., $B' \rightarrow \beta_m B'$.

Exemplo

- Considere a gramática abaixo:
  - `E → E + T | T`
    `T → num`
- Há recursão imediata à esquerda no não-terminal `E`.
- Produção sem recursão: {`E → T`}.
- Produção com recursão: {`E → E + T`}. Será eliminada da gramática.
- α_1 = `T`, β_1 = `+ T`
- Incluir a produção `E → T E'`
- Incluir a produção `E' → + T | + T E'`
- Nova gramática equivalente:
  - `E → T E' | T`
    `E' → + T | + T E'`
    `T → num`

Algoritmo geral

- Há um algoritmo geral para eliminação de recursividade à esquerda, tanto imediatas como não-imediatas.
- Há duas condições para que ele possa ser aplicado:
  - A gramática não pode ter produções com lado direito igual a ε
    - A única exceção permitida é com o símbolo inicial S, desde que ele não apareça do lado direito de nenhuma produção
  - A gramática não pode ter ciclos, ou seja, derivações da forma $\mathbf{A} \Rightarrow^{+} \mathbf{A}$
- Algoritmo:

```
Sejam os não-terminais: A1, A2, ..., An
for (i=1; i<=n; i++) {
     for (j=1; j<i; j++) {
          Sejam as produções atuais de Aj: Aj → δ1 | δ2 | ... | δm
          Substituir cada produção Ai → Ajγ pelas seguintes produções:
               Ai → δ1γ | δ2γ | ... | δmγ
     }
     Eliminar as recursividades imediatas nas produções de Ai
}
```

Exemplo

- Considere a gramática abaixo:
  - $A_1 \rightarrow A_1a \mid A_2b \mid c$
    $A_2 \rightarrow A_1d \mid A_3e \mid f$
    $A_3 \rightarrow A_1g \mid A_3h \mid i$
- Com i=1, são eliminadas as recursidades imediatas de A_1:
  - $A_1 \rightarrow A_2bX \mid cX$
    $X \rightarrow aX \mid \varepsilon$
- Quando i=2 e j=1, são consideradas as produções $A_2 \rightarrow A_1\gamma$:
  - $A_2 \rightarrow A_2bXd \mid cXd \mid A_3e \mid f$
- Com i=2, são eliminadas as recursidades imediatas de A_2:
  - $A_2 \rightarrow cXdY \mid A_3eY \mid fY$
    $Y \rightarrow bXdY \mid \varepsilon$
- Quando i=3 e j=1, são consideradas as produções $A_3 \rightarrow A_1\gamma$:
  - $A_3 \rightarrow A_2bXg \mid cXg \mid A_3h \mid i$
- Quando i=3 e j=2, são consideradas as produções $A_3 \rightarrow A_2\gamma$:
  - $A_3 \rightarrow cXdYbXg \mid A_3eYbXg \mid fYbXg \mid cXg \mid A_3h \mid i$
- Com i=3, são eliminadas as recursidades imediatas de A_3:
  - $A_3 \rightarrow cXdYbXgZ \mid fYbXgZ \mid cXgZ \mid iZ$
    $Z \rightarrow eYbXgZ \mid hZ \mid \varepsilon$

Gramática equivalente:

$A_1 \rightarrow A_2bX \mid cX$
$X \rightarrow aX \mid \varepsilon$
$A_2 \rightarrow cXdY \mid A_3eY \mid fY$
$Y \rightarrow bXdY \mid \varepsilon$
$A_3 \rightarrow cXdYbXgZ \mid fYbXgZ \mid cXgZ \mid iZ$
$Z \rightarrow eYbXgZ \mid hZ \mid \varepsilon$

Podem permanecer ou surgir recursividades à direita, que não atrapalham o *parser* LL(1)

Fatoração à esquerda

- Fatoração à esquerda é uma técnica para garantir que a tabela preditiva tenha uma única produção gramatical em cada posição.
- Este problema ocorre quando duas ou mais produções compartilham um prefixo comum. Exemplo: $A \rightarrow \alpha\beta \mid \alpha\gamma$.
- Outro exemplo clássico: `S → IF E THEN S ELSE S | IF E THEN S`
  - Como o terminal `IF` inicia duas produções, esta gramática não é LL(1).
- A fatoração à esquerda modifica as produções dos não-terminais que apresentam este problema: identifica o maior prefixo comum entre suas produções, e cria novas produções para completar a original.
- No primeiro exemplo, as produções $A \rightarrow \alpha\beta \mid \alpha\gamma$ são reescritas como $A \rightarrow \alpha A'$ e $A' \rightarrow \beta \mid \gamma$, onde α é a cadeia compartilhada mais longa.
  - Se o primeiro símbolo de β e γ for um não-terminal, gramática pode não ser LL(1).
- No exemplo clássico acima:
  - `S → IF E THEN S X`
    `X → ELSE S | ε`

Exemplos de fatoração à esquerda

- Sequência de declarações:
  - `decl-seq → decl ; decl-seq | decl`
    `decl → s`
  - `decl-seq → decl decl-seq2`
    `decl-seq2 → ; decl-seq | ε`
    `decl → s`
- Declarações que começam com um identificador:
  - `decl → identif := exp | identif ( exp-lista ) | outra`
  - `decl → identif decl2 | outra`
    `decl2 → := exp | ( exp-lista )`
- Expressões matemáticas:
  - `exp → termo exp2 | termo`
    `exp2 → + termo | - termo | + termo exp2 | - termo exp2`
    `termo → fator termo2 | fator`
    `termo2 → * fator | / fator | * fator termo2 | / fator termo2`
    `fator → ( exp ) | id | num`
  - `exp → termo exp2`
    `exp2 → + termo exp2 | - termo exp2 | ε`
    `termo → fator termo2`
    `termo2 → * fator termo2 | / fator termo2 | ε`
    `fator → ( exp ) | id | num`

Construção de tabelas preditivas

- Na implementação de um *parser* LL(1), vimos que é necessária uma tabela preditiva M.
- Dados X ∈ N e **a** ∈ T, M[X,**a**] indica a produção gramatical de X que será utilizada no seu reconhecimento quando **a** for o próximo *token* de entrada.
- Na montagem da tabela M, utilizaremos duas funções auxiliares:
  - First(α): retorna o conjunto de todos os terminais que podem aparecer no início das cadeias derivadas a partir da forma sentencial α. Se $\alpha \Rightarrow^* \varepsilon$, então ε também estará neste conjunto.
  - Follow(X): retorna o conjunto de todos os terminais que podem suceder o não-terminal X em alguma forma sentencial válida. Caso X seja o símbolo mais à direita em alguma forma sentencial, então $ (símbolo de fim de entrada) também estará neste conjunto.
- A seguir, vamos mostrar como estas funções são calculadas.

Pseudocódigo de First

```
set First(α) {                                  // α ∈ (N ∪ T)*
    if ((α == ε) || (α ∈ T)) then return {α};
    if (α ∈ N) then {
       Sejam α → α1 | α2 | ... | αn todas as produções de α;
       return First(α1) ∪ First(α2) ∪ ... ∪ First(αn);
    }
    Seja α = X1X2...Xk, onde Xi ∈ (N ∪ T);
    F = φ;
    i = 0;
    repeat
       i++;
       F = F ∪ (First(Xi)-{ε});
    until ((Xi ∈ T) || ((Xi ∈ N) && (ε ∉ First(Xi))));
    if ((i == k) && (ε ∈ First(Xk)) then F = F ∪ {ε};
    return F;
}
```

- Como há diversas chamadas recursivas com eventuais *loops* infinitos (recursão à esquerda, por exemplo), o cálculo é feito do seguinte modo:
  - Calcula-se esta função para cada produção gramatical, armazenando os resultados numa tabela no estilo *Programação Dinâmica*.
  - Percorre-se esta tabela até que ela deixe de ser modificada.

Exemplo 1

- Considere a gramática abaixo:
  - T = {(,), +, -, *, NUM}
  - exp → exp op1 termo | termo
    op1 → + | -
    termo → termo op2 fator | fator
    op2 → *
    fator → (exp) | NUM
- Tabela com resultados de First:

Produções gramaticais	Percurso 1	Percurso 2	Percurso 3
exp → exp op1 termo			
exp → termo			First(exp) = {(,NUM}
op1 → +	First(op1) = {+}		
op1 → -	First(op1) = {+,-}		
termo → termo op2 fator			
termo → fator		First(termo) = {(,NUM}	
op2 → *	First(op2) = {*}		
fator → (exp)	First(fator) = {(}		
fator → NUM	First(fator) = {(,NUM}		

Exemplo 2

- Considere a gramática abaixo:
  - T = {0, 1, (,), **IF**, **ELSE**, **OUTRO**}
  - **stat** → **ifstat** | **OUTRO**
    ifstat → **IF** (**exp**) **stat elsepart**
    elsepart → **ELSE stat** | ε
    exp → **0** | **1**
- Tabela com resultados de First:

Produções gramaticais	Percurso 1	Percurso 2
stat → ifstat		FIRST(stat) = {IF,OUTRO}
stat → OUTRO	FIRST(stat) = {OUTRO}	
ifstat → IF (exp) stat elsepart	FIRST(ifstat) = {IF}	
elsepart → ELSE stat	FIRST(elsepart) = {ELSE}	
elsepart → ε	FIRST(elsepart) = {ELSE,ε}	
exp → 0	FIRST(exp) = {0}	
exp → 1	FIRST(exp) = {0,1}	

Pseudocódigo de Follow

```
BuildFollow() {          // Cria a tabela Follow indexada por não-terminais
   Follow(S) = {$};      // S é o símbolo inicial
   for each A ∈ N-{S} do
      Follow(A) = φ;
   for each A → X1X2...Xn do          // Fase 1: inclui First calculados
       for each Xi ∈ N do
           if (Xi+1...Xn != ε) then
               Follow(Xi) = Follow(Xi) ∪ (First(Xi+1...Xn)-{ε});
   repeat                             // Fase 2: inclui Follow calculados
      for each A → X1X2...Xn do {
          i = n+1;
          repeat
             i--;
             if (Xi ∈ N) then Follow(Xi) = Follow(Xi) ∪ Follow(A);
          until ((Xi ∉ N) || (ε ∉ First(Xi));
      }
   until (não haja modificações na tabela Follow);
}
```

- Como pode ser observado, Follow depende do cálculo prévio de First.
- Mais adiante, na montagem da tabela preditiva, veremos que Follow será necessário apenas quando algum First contiver ε.

Exemplo 1

- Considere novamente a gramática abaixo:
  - T = {(,), +, -, *, NUM}
  - exp → exp op1 termo | termo
    op1 → + | -
    termo → termo op2 fator | fator
    op2 → *
    fator → (exp) | NUM

First(exp) = {(,NUM}
First(termo) = {(,NUM}
First(fator) = {(,NUM}
First(op1) = {+,-}
First(op2) = {*}

- Embora não seja necessário neste caso, vamos simular o cálculo de Follow:

Produções gramaticais	Fase 1	Fase 2
exp → exp op1 termo	Follow(exp) = {$,+,-} Follow(op1) = {(,NUM}	Follow(termo) = {$,+,-,),*}
exp → termo		
termo → termo op2 fator	Follow(termo) = {*} Follow(op2) = {(,NUM}	Follow(fator) = {$,+,-,),*}
termo → fator		
fator → (exp)	Follow(exp) = {$,+,-,)}	

Exemplo 2

- Considere novamente a gramática abaixo:
  - T = {0, 1, (,), **IF**, **ELSE**, **OUTRO**}
  - **stat** → **ifstat** | **OUTRO**
    ifstat → **IF** (**exp**) **stat elsepart**
    elsepart → **ELSE stat** | ε
    exp → **0** | **1**

First(stat) = {IF,OUTRO}
First(ifstat) = {IF}
First(elsepart) = {ELSE,ε}
First(exp) = {0,1}

- Simulação do cálculo de Follow:

Produções gramaticais	Fase 1	Fase 2
stat → ifstat	Follow(stat) = {$}	Follow(ifstat) = {$,ELSE}
ifstat → IF (exp) stat elsepart	Follow(exp) = {)} Follow(stat) = {$,ELSE}	Follow(elsepart) = {$,ELSE}
elsepart → ELSE stat		

Construção da tabela preditiva LL(1)

- Para cada não-terminal A com produção $A \rightarrow \alpha$:
  - Para cada terminal **a** de First(α), incluir $A \rightarrow \alpha$ em M[A,**a**]
    - Há produção $A \rightarrow \alpha$ e existe derivação $\alpha \Rightarrow^* \mathbf{a}\beta$
  - Se $\varepsilon \in$ First(α), para cada terminal **a** de Follow(α), incluir $A \rightarrow \alpha$ em M[A,**a**]
    - Há produção $A \rightarrow \alpha$ e existem derivações $S \Rightarrow^* \beta A \mathbf{a} \gamma$ e $\alpha \Rightarrow^* \varepsilon$

Exemplo

- Considere novamente a gramática abaixo:
  - T = {0, 1, (,), **IF**, **ELSE**, **OUTRA**}
  - **stat** → **ifstat** | **OUTRO**
    ifstat → **IF** (**exp**) **stat elsepart**
    elsepart → **ELSE stat** | ε
    exp → 0 | 1

First(stat) = {IF,OUTRO}
First(ifstat) = {IF}
First(elsepart) = {ELSE,ε}
First(exp) = {0,1}

Follow(stat) = {$,ELSE}
Follow(ifstat) = {$,ELSE}
Follow(elsepart) = {$,ELSE}
Follow(exp) = {)}

- Montagem da tabela preditiva M:

M(N,T)	OUTRO	IF	ELSE	(	)	0	1	$
stat	stat → OUTRO	stat → ifstat						
ifstat		ifstat → IF (exp) stat elsepart						
elsepart			elsepart → ELSE stat elsepart → ε					elsepart → ε
exp						exp → 0	exp → 1	

- Há duas produções na posição M[**elsepart**, **ELSE**], devido à ambiguidade do **ELSE** pendente.
- A ambiguidade pode ser resolvida priorizando uma produção.

Recuperação de erros

- Uma das tarefas da análise sintática é detectar eventuais presenças de erros:
  - Dar uma mensagem adequada, indicando onde ocorreu;
  - Corrigir erros simples, como falta de pontuação;
  - Recuperar-se para prosseguir na análise.
- Aspectos importantes:
  - Determinar a ocorrência de um erro o mais cedo possível;
  - Analisar o máximo possível, antes de encerrar o processo;
  - Evitar cascata de erros e de mensagens (laços infinitos).

Recuperação nos *parsers* recursivos

- A forma padrão é o "modo pânico": consome vários tokens, tentando encontrar algum ponto para continuar ou encerrar a análise.
- Deste modo, evita-se o risco do *loop* infinito.
- Mecanismo básico de implementação
  - A cada procedimento de análise, é associado um parâmetro extra, indicando um conjunto de *tokens* de sincronização.
  - Diante de um erro, o *parser* ignora a entrada até encontrar um desses *tokens*.
  - Os conjuntos definidos pela função Follow são candidatos naturais para esta sincronização.
  - Também podem ser utilizados os conjuntos First, para evitar o descarte de novas produções gramaticais.

Recuperação nos *parsers* LL(1)

- Um erro primário ocorre quando um não-terminal *A* está no topo da pilha e o *token* de entrada é **a**, mas não há produções em M[*A*,**a**].
- Possíveis soluções:
  - Retirar *A* da pilha (ação **pop**). Boa opção quando o *token* corrente for **$** ou estiver em Follow(*A*);
  - Consumir *tokens* da entrada até encontrar um que permita reinício da análise (ação **skip**). Boa opção quando o *token* corrente não for **$** e nem estiver em First(*A*) U Follow(*A*);
  - Empilhar um novo não-terminal. Somente em situações excepcionais. Por exemplo: quando a pilha ficar vazia e a entrada não. Neste caso, empilha-se o símbolo inicial S e skip entrada até encontrar um *token* pertencente a First(S).
- As ações **pop** e **skip** são inseridas nas posições vagas de M.

Exemplo

- Considere a gramática abaixo:
  - exp → termo exp2
    exp2 → op1 termo exp2 | ε
    op1 → + | -
    termo → fator termo2
    termo2 → op2 fator termo2 | ε
    op2 → *
    fator → (exp) | NUM
- Tabela preditiva com recuperação de erros:

M[N,T]	(	NUM	)	+	-	*	$
exp	exp → termo exp2	exp → termo exp2	pop	skip	skip	skip	pop
exp2	skip	skip	exp2 → ε	exp2 → op1 termo exp2	exp2 → op1 termo exp2	skip	exp2 → ε
op1	pop	pop	skip	op1 → +	op1 → -	skip	pop
termo	termo → fator termo2	termo → fator termo2	pop	pop	pop	skip	pop
termo2	skip	skip	termo2 → ε	termo2 → ε	termo2 → ε	termo2 → op2 fator termo2	termo2 → ε
op2	pop	pop	skip	skip	skip	op2 → *	pop
fator	fator → (exp)	fator → NUM	pop	pop	pop	pop	pop

Exemplo de análise

- Considere a tabela preditiva anterior:

M[N,T]	(	NUM	)	+	-	*	$
exp	exp → termo exp2	exp → termo exp2	pop	skip	skip	skip	pop
exp2	skip	skip	exp2 → ε	exp2 → op1 termo exp2	exp2 → op1 termo exp2	skip	exp2 → ε
op1	pop	pop	skip	op1 → +	op1 → -	skip	pop
termo	termo → fator termo2	termo → fator termo2	pop	pop	pop	skip	pop
termo2	skip	skip	termo2 → ε	termo2 → ε	termo2 → ε	termo2 → op2 fator termo2	termo2 → ε
op2	pop	pop	skip	skip	skip	op2 → *	pop
fator	fator → (exp)	fator → NUM	pop	pop	pop	pop	pop

- Simulação da análise de **(NUM + *)** :

Pilha	Entrada	Ações
$ exp	(NUM *) $	exp → termo exp2
$ exp2 termo	(NUM *) $	termo → fator termo2
$ exp2 termo2 fator	(NUM *) $	fator → (exp)
$ exp2 termo2) exp (	(NUM *) $	*token* (
$ exp2 termo2) exp	NUM *) $	exp → termo exp2
$ exp2 termo2) exp2 termo	NUM *) $	termo → fator termo2
$ exp2 termo2) exp2 termo2 fator	NUM *) $	fator → NUM

Exemplo de análise: continuação

M[N,T]	(	NUM	)	+	-	*	$
exp	exp → termo exp2	exp → termo exp2	pop	skip	skip	skip	pop
exp2	skip	skip	exp2 → ε	exp2 → op1 termo exp2	exp2 → op1 termo exp2	skip	exp2 → ε
op1	pop	pop	skip	op1 → +	op1 → -	skip	pop
termo	termo → fator termo2	termo → fator termo2	pop	pop	pop	skip	pop
termo2	skip	skip	termo2 → ε	termo2 → ε	termo2 → ε	termo2 → op2 fator termo2	termo2 → ε
op2	pop	pop	skip	skip	skip	op2 → *	pop
fator	fator → (exp)	fator → NUM	pop	pop	pop	pop	pop

Pilha	Entrada	Ações
$ exp2 termo2) exp2 termo2 NUM	NUM *) $	*token* NUM
$ exp2 termo2) exp2 termo	*) $	skip
$ exp2 termo2) exp2 termo	) $	pop
$ exp2 termo2) exp2	) $	exp2 → ε
$ exp2 termo2)	) $	*token*)
$ exp2 termo2	$	termo2 → ε
$ exp2	$	exp2 → ε
$	$	aceitação