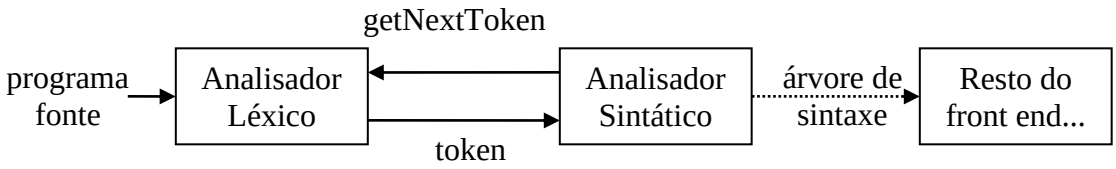
Análise Sintática

Introdução

Analisador sintático:

- obtém tokens do analisador léxico;
- verifica se a sequência de tokens compõe uma forma sentencial válida para a gramática da linguagem, emitindo mensagens de erro, caso necessário;
- constrói uma árvore de sintaxe para o programa de entrada.



- Exemplo:

Lexemes: `num` : números inteiros
`id` : identificadores
outros símbolos: `+ * ( )`

Gramática G1: `E → E + E | E * E | (E) | num | id`
-símbolos terminais:
`+ * ( ) num id`

A gramática G1 permite derivação de formas sentenciais como

`ab + 123`
`E => E + E => id + E => id + num`
(`ab` é identificado como o terminal `id` pelo analisador léxico, e `123` é associado a `num`)

`a + b * c`
`E => E + E => id + E => id + E * E => id + id * E => id + id * id`

Mas G1 é ambígua. A forma sentencial `a + b * c` também pode ser gerada com
`E => E * E => E + E * E => id + E * E => id + id * E`
`  => id + id * id`

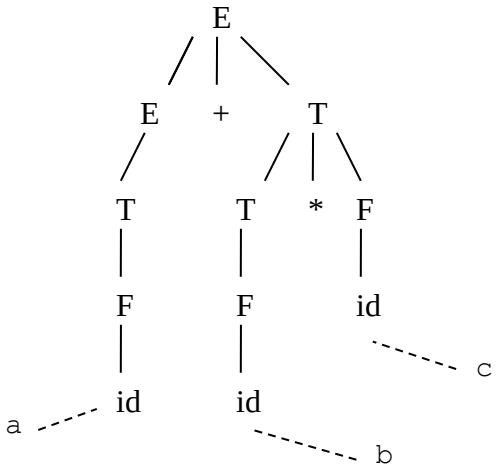
Gramáticas ambíguas, em geral, não são adequadas para definir o comportamento de analisadores sintáticos, pois podem associar uma mesma entrada a mais de uma árvore sintática.

A Gramática G2 abaixo representa a mesma linguagem de G1, mas não é ambígua.

Gramática G2:

```
E → E + T | T
T → T * F | F
F → (E) | id | num
```

Única árvore sintática para `a + b * c`:



A gramática G2 produz árvores sintáticas que refletem a prioridade do operador `*` sobre o operador `+` nas expressões, podendo facilitar o processamento dessas operações em fases posteriores de um compilador. As operações são mais facilmente processadas em uma árvore se um operador e seus operandos estiverem reunidos em uma mesma subárvore. Observe, na árvore acima, que os operandos `b` e `c` estão em uma mesma subárvore que o operador *, indicando que pode-se processar prioritariamente de forma simples a operação `(b*c)`, usando algoritmos recursivos sobre a árvore. O processamento da operação `+` poderia ser realizado em uma etapa posterior, tendo como operandos `a` e `(b*c)`.

Abordagens para construção de analisadores sintáticos

Dada uma gramática, pode-se construir um analisador sintático usando diferentes técnicas.

Tipos de analisador sintático:

- Universal: pode analisar qualquer gramática. Em geral, não são métodos muito eficientes.
- Top-down: constroem árvores sintáticas a partir da raiz para as folhas.
- Bottom-up: constroem árvores sintáticas a partir das folhas até atingir a raiz.

Os analisadores sintáticos top-down e bottom-up mais eficientes funcionam apenas para subclasses de gramáticas livres de contexto. Para atingir eficiência, essas técnicas impõem restrições sobre a forma como as produções podem ser expressas, definindo, por exemplo, classes de gramáticas conhecidas LL e LR. Mas essas classes de gramáticas são poderosas o suficiente para expressar a maioria das principais construções de linguagens de programação modernas.

Análise Sintática Top-Down

Objetivo: construir a árvore de derivação para uma entrada, partindo do símbolo inicial da gramática. As derivações utilizadas são sempre mais-à-esquerda.

Em cada passo de uma análise top-down, a tarefa principal é decidir qual produção a ser aplicada ao terminal mais-à-esquerda. À medida que prefixos terminais vão sendo gerados, eles são "casados" com a entrada e mais símbolos da entrada podem ser lidos.

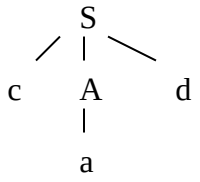
Exemplo:

Gramática G3:

```
S → cAd
A → b | a
```

Para construir a árvore de derivação associada à entrada w = **cad**:

```
S => cAd    (usando S → cAd)
  => cbd    (usando A → b)
  ... verifica que não casa com w
  => cAd    (retorna ao estado anterior)
  => cad    (usando A → a)
  OK - entrada gerada!
```



O exemplo acima apresenta um caso de uso de "backtracking". Uma produção escolhida em um ponto do processo levou a uma situação de erro. Uma configuração anterior é recuperada e uma produção alternativa é aplicada, levando ao resultado desejado.

Analisadores Sintáticos Preditivos

O uso de backtracking não é desejável em analisadores sintáticos por causa do gasto adicional de tempo. Para linguagens de programação comuns, em geral é possível construir analisadores top-down sem backtracking, ou seja, a cada passo, é possível determinar exatamente a única produção a ser aplicada. Analisadores sintáticos top-down que dispensam backtracking são chamados *preditivos*.

Para decidir qual produção utilizar em cada passo, pode-se analisar quais são os próximos símbolos da entrada, e compará-los com o que pode ser gerado pelas produções. A essa parte da entrada que é verificada a cada passo da análise sintática, chamamos de *lookahead*. Quanto menor o número de símbolos que devem ser verificados, mais eficiente pode ficar o processo. Por exemplo, se é suficiente analisar apenas 1 símbolo da entrada, dizemos que o analisador funciona com *lookahead* de 1 símbolo.

A gramática G3 permite construir um analisador top-down preditivo com *lookahead* de 1 símbolo. Se o próximo símbolo da entrada é `a`, a produção a ser usada deve ser `A → a`. Se o próximo símbolo da entrada é `b`, a produção a ser usada deve ser `A → b`.

Restrições sobre Gramáticas para Análise Top-Down

O exemplo com a gramática G3 apresenta uma restrição sobre o formato das produções de uma gramática, caso essa seja usada para construir analisadores top-down preditivos. A abordagem top-down apresenta outras restrições, e uma das mais importantes é a não possibilidade de usar recursividade à esquerda. Para entender essa restrição, vamos analisar a gramática G2 novamente.

```
E → E + T | T
T → T * F | F
F → (E) | id | num
```

Suponha que a entrada seja uma expressão que comece com "`(`", `id` ou `num`. O símbolo inicial é `E`. A partir desse símbolo, pode-se escolher iniciar a derivação usando duas produções, `E → E + T` e `E → T`. Mas ambas opções podem levar a uma forma que inicia com um dos símbolos mencionados. Por exemplo:

```
E => T => F => id
```

ou

```
E => E + T => T + T => F + T => id + T => ...
```

Em ambas as derivações, a forma gerada iniciará com `id`. Assim, não se pode decidir qual produção usar observando apenas o primeiro símbolo da entrada (*lookahead* de 1 símbolo).

O problema apresentado acima é ainda mais grave. Pode-se demonstrar que há casos em que é impossível determinar qual produção a ser utilizada, não importando o número (finito) de símbolos da entrada investigados. O problema da gramática G2 é que ela é recursiva à esquerda. A aplicação da produção `E → E + T`, por exemplo, pode gerar indefinidamente uma sentença que inicia com `E`, que a qualquer momento pode ser trocada por `T`.

Observe um caso patológico:
`(((((((((1)))))))))+2`

Na entrada acima, a primeira produção a ser utilizada deveria ser `E → E + T`. O primeiro `E` à direita irá depois derivar toda a expressão entre os vários parêntesis. Mas para descobrir isso, seria necessário ler todos os símbolos da entrada até encontrar +. Na entrada

`((((((((((1))))))))))`
por outro lado, a primeira produção a ser utilizada deveria ser `E → T`.

Conclusão:

Gramáticas com recursividade à esquerda não são adequadas para produzir analisadores léxicos preditivos com abordagem top-down.

Transformações sobre Gramáticas

A conclusão apresentada logo acima indica que não é possível definir um analisador sintático preditivo com abordagem top-down que reconheça a a linguagem L(G2)? Como superar essa restrição?

Sabe-se que, para qualquer gramática livre de contexto com recursividade à esquerda, sempre é possível construir uma gramática equivalente sem recursão à esquerda. Observe a gramática G4 a seguir.

Gramática G4:

```
E  → T E'
E' → + T E' | λ
T  → F T'
T' → * F T' | λ
F → (E) | id | num
```

Podemos afirmar que L(G4) = L(G2). As produções recursivas à esquerda foram trocadas por produções equivalentes, com recursividade à direita.

Usando a gramática G4, pode-se sempre decidir qual produção a ser utilizada, verificando apenas um símbolo da entrada restante (lookahead de 1 símbolo). Por exemplo, quando o símbolo não terminal a ser processado é `E'`, a produção `E' → + T E'` será usada se o próximo símbolo da entrada for `+`; caso contrário, (exceto para os casos de erro de sintaxe) a produção a ser usada deve ser `E' → λ`. Mais tarde, apresentaremos formas de se identificar exatamente os símbolos da entrada que podem ser válidos quando uma produção `λ` é usada.

Estendendo a Notação

Comparando as gramáticas equivalentes G2 e G4, podemos afirmar que G2 apresenta mais claramente a intenção do projetista para a sintaxe da linguagem. Transformações introduzidas em G4 serviram para evitar a recursividade à esquerda.

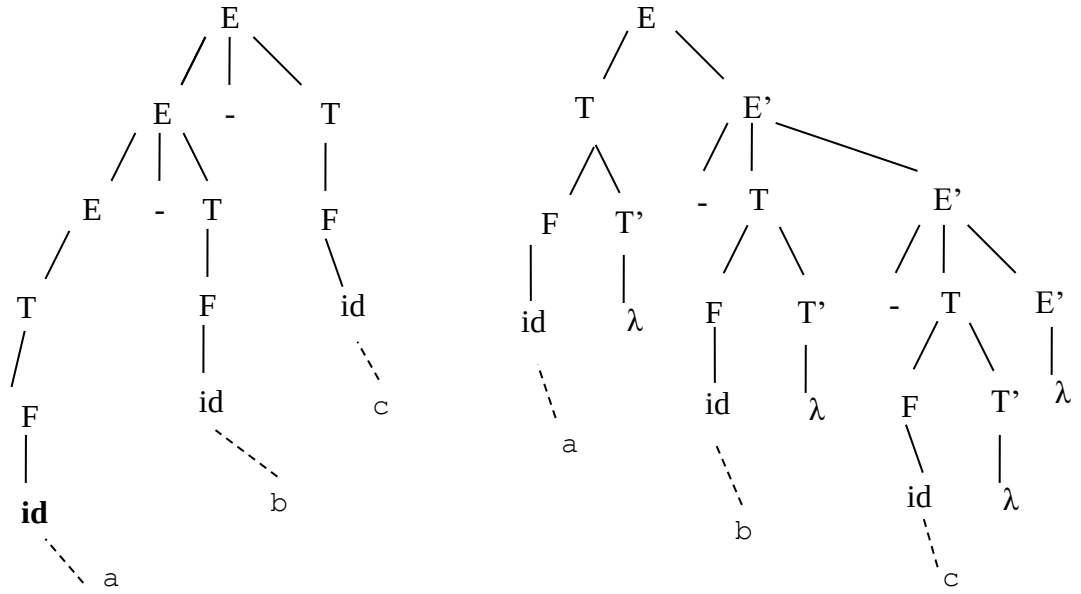
Além disso, suponha uma extensão da linguagem de expressões, considerando também operador para subtração:

Gramática G2':	**Gramática G4':**
E → E + T | E - T | T T → T * F | F F → (E) | id | num	E → T E' E' → + T E' | - T E' | λ T → F T' T' → * F T' | / F T' |λ F → (E) | id | num

Uma expressão como `a-b-c` deve ser interpretada de modo que a primeira operação de subtração seja processada antes, sendo equivalente a `(a-b)-c`. Dizemos que o operador − (e também + e *) são ***associativos à esquerda***. Por razões que ficarão mais claras mais tarde, é interessante que a árvore de sintaxe gerada pela gramática reflita a correta precedência e associatividade dos operadores - isso poderá facilitar mais tarde o processamento da árvore.

Observe as árvores de sintaxe para a forma sentencial `a-b-c`.

Gramática G2': **Gramática G4':**



Na árvore produzida pela gramática G2', fica claro que a subexpressão `a-b` está representada em uma subárvore própria, separada do resto do expressão. Isso pode facilitar o processamento dessa subexpressão separadamente. Essa distribuição não pode ser diretamente observada na árvore produzida pela gramática G4'.

Para melhorar a clareza e possibilitar a construção de árvores mais adequadas, frequentemente ferramentas que trabalham com análise sintática top-down utilizam uma extensão da notação de gramáticas livre de contexto. O exemplo a seguir aplica notação similar à usada pela ferramenta ANTLR, que mescla operadores de expressões regulares com a notação de gramáticas livre de contexto.

Na gramática G5 abaixo, adotamos uma convenção diferente para identificar os símbolos terminais, não terminais e quais são os meta-símbolos usados pelo modelo.
Os não terminais são letras maiúsculas; neste caso, `E`, `T` e `F`, sendo `E` o símbolo inicial.
Os símbolos terminais são escritos em negrito; neste caso: **`+ - * / ( ) id num`**.
São usados como meta-símbolos:

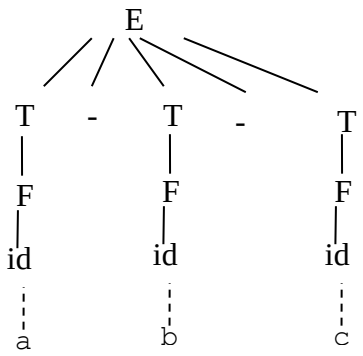
→	Para separar o lado esquerdo do lado direito de uma produção.
|	Para indicar mais de um lado direito para um mesmo lado esquerdo, como nas produções do não terminal E. Também para representar escolha entre opções, similar a expressões regulares, como no lado direito das produções de E e T.
*	Para indicar repetição, como em expressões regulares.
( )	Para indicar precedência entre os outros meta-símbolos.

Gramática G5:

`E → T ((`**`+`** `|` **`-`**`) T)*`
`T → F ((`**`*`** `|` **`/`**`) F)*`
`F →` **`(`**`E`**`)`** `|` **`id`** `|` **`num`**

Observe que o símbolo * tem dois usos diferentes. É usado como um símbolo terminal, podendo fazer parte das expressões geradas, como em `a*2`. Também é usado para denotar operador de repetição, como em expressões regulares. Por exemplo, a primeira produção é interpretada como: `E` produz `T`, seguido de qualquer número de repetições de `+T` ou `-T`. Os parêntesis também são usados com dois significados diferentes, podendo aparecer como terminais ou usados para indicar precedência entre os meta-operadores.

A gramática G5 oferece uma especificação para a linguagem de expressões que pode ser considerada mais clara que G4, sem ter que recorrer ao artifício de recursividade à esquerda. A figura abaixo apresenta uma árvore de sintaxe produzida pela gramática G5, quando a forma sentencial `a-b-c` é gerada.



Conjuntos FIRST e FOLLOW

Vimos que analisadores sintáticos top-down que dispensam backtracking são chamados *analisadores preditivos*.

Para definir precisamente um conjunto de gramáticas para as quais é possível construir analisadores sintáticos preditivos, vamos usar os conceitos FIRST e FOLLOW, apresentados a seguir.

Conjunto FIRST - motivação

Para uma palavra *w* formada por símbolos de uma gramática, FIRST(*w*) é o conjunto de terminais que podem aparecer no início de palavras derivadas a partir de *w*. Adicionalmente, o símbolo λ pode estar em FIRST(*w*), se λ é derivável a partir de *w*.

Pare se ter uma ideia de como essa definição pode ser aplicada, observe a gramática a seguir:

G_1:
S → ABC | D
A → a
B → b
C → c
D → d

Suponha que a entrada a ser gerada seja abc, e que possamos usar *lookahead* de 1 símbolo (verificar apenas o próximo símbolo da entrada para decidir qual produção utilizar).

A sequência de derivações começa com o símbolo inicial S. Ao verificarmos o primeiro símbolo da entrada (a), devemos decidir qual das 2 produções de S utilizar. Podemos inferir que a produção correta deve ser S → ABC , pois precisamos derivar algo que comece com "a" e o primeiro símbolo de sequência derivada a partir de ABC é a . Por outro lado, o primeiro símbolo derivado a partir de D só pode ser d.
O conceito de FIRST nos ajuda nesse sentido porque:

- FIRST(ABC) = FIRST(A) = { a }
- FIRST(D) = { d }

Observe que aplicamos o conceito de FIRST aos lados direitos das produções, para identificar qual são os primeiros símbolos que podem produzir. Para calcular FIRST(ABC), é necessário calcular FIRST(A).

Continuando a sequência, o símbolo A só pode derivar a, que é casado com a entrada, depois aplica-se as produções de B e C:

Entrada	Forma sentencial	Produção que deve ser usada
abc	S	S → ABC
abc	=> ABC	A → a
abc	=> aBC	B → b
abc	=> abC	C → c
abc	=> abc	

Como visto acima, aplicamos o conceito de FIRST aos lados direitos das produções de S para decidir qual a produção correta a ser aplicada. Mas esse conceito pode ser aplicado ao próprio símbolo S:

FIRST(S) = FIRST(ABC) ∪ FIRST(D) = { a } ∪ { d } = { a, d }

Isso pode ser útil, por exemplo, no caso de derivação de uma entrada como bac. O primeiro símbolo da entrada é b e iniciamos a derivação a partir de S. Já neste momento, podemos ter certeza de que não será possível derivar a entrada em questão, pois o cálculo de FIRST nos mostra que é impossível derivar palavra iniciando com b a partir de S. Uma mensagem de erro pode ser automaticamente gerada usando FIRST, como:

" ERRO: símbolo a ou d esperado "

Conjunto FOLLOW - motivação

Para um símbolo não terminal *A*, FOLLOW(*A*) é o conjunto de terminais que podem aparecer imediatamente à direita de *A* em alguma forma sentencial da gramática. Nesse conjunto, pode ser incluído o símbolo *$*, que denota "fim de sequência", se *A* aparecer como símbolo mais à direita de uma forma sentencial.

Pare se ter uma ideia de como essas definições podem ser aplicadas, observe a gramática a seguir (foi acrescentada produção λ ao símbolo B da gramática G_1):

G_2:
S → ABC | D
A → a
B → b | λ
C → c
D → d

Suponha que a entrada a ser gerada seja ac, e que possamos usar *lookahead* de 1 símbolo (verificar apenas o próximo símbolo da entrada para decidir qual produção utilizar).

A sequência de derivações começa com o símbolo inicial S. Usando FIRST aplicado ao lado direito das produções de S , vimos que é possível identificar que a produção correta a ser usada é S → ABC . Continuando a sequência de derivações, chegaremos ao ponto destacado a seguir:

Entrada	Forma sentencial	Produção que deve ser usada
ac	S	S → ABC
ac	=> ABC	A → a
~~ac~~	=> aBC	B → ???

Como definir qual das duas produções associadas a B deve ser utilizada?

B → b ou B → λ

Temos que:

- FIRST(b) = { b }
- FIRST(λ) = { λ }

Logo a primeira produção não poderia ser usada, pois a partir dela não se pode gerar c. Mas o que dizer da segunda produção?

Se B produzir λ, o símbolo c deverá ser gerado por algum outro símbolo que venha logo após B na forma sentencial que está sendo derivada. Analisando os lados direitos das produções, observamos que B só aparece na produção S → ABC, e que logo após B aparece um símbolo C, que por sua vez pode gerar um símbolo c. Assim, estamos seguros que podemos usar B → λ, pois na sequência, o símbolo c poderá ser gerado. O conceito de FOLLOW nos ajuda nesse caso, pois:

FOLLOW(B) = FIRST(C) = { c }

Derivação completa:

Entrada	Forma sentencial	Produção que deve ser usada
ac	S	S → ABC
ac	=> ABC	A → a
~~ac~~	=> aBC	B → λ
~~ac~~	=> aC	C → c
~~ac~~	=> ac	

Cálculo de FOLLOW em outras situações

Em geral, para se definir quais produções aplicar, pode ser necessário calcular FIRST para cada símbolo não terminal, e depois usar esses valores para calcular FIRST para os lados direitos das produções e para calcular FOLLOW para os símbolos não terminais.

Na gramática G_2, para calcular FOLLOW(B), usamos FIRST(C) porque C aparece logo após B em um lado direito de uma produção. Mas como fazer para calcular FOLLOW de um símbolo se ele aparece como último símbolo do lado direito de uma produção?
O exemplo a seguir apresenta um caso como esse.

G_3:
S → AE
A → CD
C → c
D → d | λ
E → e

Suponha que se deseje derivar uma entrada ce :

S => AE => CDE => cDE => ?

O próximo passo consiste em decidir qual produção de D usar. Certamente não poderá ser a produção D → d , pois o próximo símbolo da entrada (lookahead) é e . Uma alternativa seria usar a produção D → λ , mas para iss temos que saber se, após D, o símbolo e poderá ser gerado. Ou seja, temos que calcular FOLLOW(D) e verificar se contém e .

Como D só aparece no lado direito da produção A → CD , não temos como verificar diretamente nessa produção quais símbolos poderiam ser gerados após D . Assim, os símbolos que podem aparecer logo após D são exatamente aqueles que podem aparecer após A. Ou seja:

FOLLOW(D)
= FOLLOW(A)
(porque D aparece no final da produção A → CD)
= FIRST(E)
(porque E aparece logo após A na produção S → AE)
= { e }

Sendo assim, é seguro usar a produção D → λ para o caso em que o próximo símbolo da entrada é e . A sequência de derivações completa seria:

Entrada	Forma sentencial	Produção que deve ser usada
ce	S	S → AE
ce	=> AE	A → CD
ce	=> CDE	C → c
~~c~~e	=> cDE	D → λ
~~c~~e	=> cE	E → e
~~ce~~	=> ce	

Procedimentos gerais para cálculo de FIRST e FOLLOW

Considere uma gramática livre de contexto G = (V, ∑, P, S).

Cálculo de FIRST(*X*), para qualquer X ϵ $(V \cup \sum)^*$:

- se *X* é apenas um terminal, então FIRST(*X*) = { *X* };
- se *X* é apenas um não-terminal e existe produção $X \rightarrow w$, então FIRST(*w*) está incluído em FIRST(*X*);
- se *X* é $Y_1Y_2...Y_n$, e $\lambda \in$ FIRST(Y_i) para todo i = 0, ..., k-1 (k < n), então todo elemento de FIRST(Y_k) está incluído em FIRST(*X*); caso $\lambda \in$ FIRST(Y_i) para todo i = 1, ..., n , então λ está incluído em FIRST(*X*).

Cálculo de FOLLOW(*A*), onde *A* é um não terminal da gramática:

1. se *A* é o símbolo inicial da gramática, então FOLLOW(*A*) contém $ (lembre que esse símbolo é usado para denotar "fim de arquivo");
2. se existe produção $B \rightarrow uAv$, então todos os elementos de FIRST(*v*), exceto λ, estão incluídos em FOLLOW (*A*);
3. se existe produção $B \rightarrow uA$ ou uma produção $B \rightarrow uAv$ onde FIRST(*v*) contém λ, então todos os elementos de FOLLOW(*B*) estão incluídos em FOLLOW(*A*).

Vamos mostrar esses conceitos sendo aplicados na seguinte gramática:

```
E  → T E'
E' → + T E' | λ
T  → F T'
T' → * F T' | λ
F → (E) | id | num
```

FIRST(E) = FIRST(T) = FIRST(F)
= { (, id , num }
FIRST(E') = { + , λ }
FIRST(T') = { * , λ }

FOLLOW(E) = { $,) }
(usando as definições 1 e 2 de FOLLOW)

FOLLOW(E') = FOLLOW(E) = { $,) }
(definição 3)

FOLLOW(T) =
FIRST(E') (definição 2, exceto λ)
união com
FOLLOW(E')
(definição 3; T acaba sendo o último símbolo em E' → + T E')
= { $, + ,) }

FOLLOW(T') = FOLLOW(T) = { $, + ,) }
(definição 3)

FOLLOW(F) =
FIRST(T') (definição 2, exceto λ)
união com
FOLLOW(T)
(definição 3; F acaba sendo o último símbolo em T → F T')
união com
FOLLOW(T')
(definição 3; F acaba sendo o último símbolo em T' → * F T')
= { * , $, + ,) }

Gramáticas LL(k)

Os conceitos de FIRST e FOLLOW são usados a seguir para definir a classe de gramáticas conhecida como Gramáticas LL(k).

Para a classe de gramáticas chamada LL(k), sempre é possível construir analisadores sintáticos preditivos.
L : Left to rigth (leitura da entrada da esquerda para a direita)
L : Leftmost derivation (derivação mais-à-esquerda)
k : para decidir qual produção usar, *k* símbolos da entrada são usados como lookahead

Uma linguagem é LL(k) se for possível construir uma gramática LL(k) que gere essa linguagem. Nenhuma gramática ambígua ou com recursividade à esquerda é LL(k).

Para uma gramática G ser LL(1), ou seja, usando apenas 1 símbolo da entrada como lookahead, sempre que existirem 2 produções `A → u | v`, onde u e v são palavras formadas por terminais e não terminais, as seguintes condições devem ser satisfeitas:

1. u nunca deriva uma palavra começando com terminal *a* se v derivar palavra que comece com esse mesmo terminal (e vice-versa);
2. u e v não derivam ambos a palavra nula;
3. se u deriva a palavra nula, então v não deriva nenhuma palavra que comece com um símbolo de FOLLOW(A) (e vice-versa).

O algoritmo a seguir mostra como uma tabela *M* para analisador sintático preditivo pode ser construída, para uma gramática LL(1). A tabela é uma matriz de 2 dimensões: uma linha para cada símbolo não terminal da gramática, e uma coluna para cada símbolo terminal da gramática (incluindo símbolo $, que representa fim da entrada).

Para cada produção $A \rightarrow u$:

1. Para cada terminal a em FIRST(u), adicione $A \rightarrow u$ a $M[A,a]$.
2. Se λ pertence a FIRST(u) (ou seja, u pode gerar λ) então para cada símbolo b em FOLLOW(A), adicione $A \rightarrow u$ a $M[A,b]$. O símbolo b pode ser inclusive $.

A tabela pode ser usada por um analisador preditivo da seguinte maneira: quando for necessário decidir qual produção a ser aplicada, a tabela é consultada na entrada $M[A,a]$, onde A é o não terminal mais-à-esquerda da forma sentencial e a é o próximo símbolo da entrada (*lookahead*).

Exemplo (gramática de expressões, sem recursividade à esquerda, simplificada):

```
E → T E′
E′→ + T E′ | λ
T → F T′
T′→ * F T′ | λ
F → ( E ) | id
```

FIRST(T E′) = { (, id }
FIRST(F T′) = { (, id }

FOLLOW(E) = {) , $ }
FOLLOW(E′) = {) , $ }
FOLLOW(T) = { + ,) , $ }
FOLLOW(T′)= { + ,) , $ }
FOLLOW(F) = { + , * ,) , $ }

Tabela para analisador preditivo:

não terminal	símbolo de entrada					
	id	+	*	(	)	$
E	E→ T E′			E→ T E′		
E′		E′→ + T E′			E′→ λ	E′→ λ
T	T→ F T′			T→ F T′		
T′		T′→ λ	T′→ * F T′		T′→ λ	T′→ λ
F	F→ id			F→ (E)		

Como não é gerado nenhum conflito na tabela, então a gramática G é LL(1).

Construção de analisadores sintáticos preditivos

Duas formas simples podem ser usadas para se construir um analisador sintático preditivo, a partir de uma tabela *M* como a apresentada na seção anterior.

Analisador preditivo não recursivo

Uma pilha armazena os símbolos ainda não utilizados das formas sentenciais. A tabela *M* construída para o analisador preditivo é usada da seguinte forma:

- Se no topo da pilha houver um símbolo não terminal, a tabela é consultada usando esse símbolo e o próximo símbolo terminal da entrada. O não terminal é substituído pelo lado direito da produção identificada.
- Se houver terminal no topo da pilha, ele deve "casar" com o próximo símbolo terminal da entrada.

Para ilustrar o procedimento, um rastreio é exibido a seguir, considerando como entrada `id + id * id`.

entrada casada	pilha	entrada restante	ação
	E $	id + id * id $	E → T E′
	T E′ $	id + id * id $	T → F T′
	F T′ E′ $	id + id * id $	F → id
	id T′ E′ $	id + id * id $	casa com id
id	T′ E′ $	+ id * id $	T′→ λ
id	E′ $	+ id * id $	E′→ + T E′
id	+ T E′ $	+ id * id $	casa com +
id +	T E′ $	id * id $	T → F T′
id +	F T′ E′ $	id * id $	F → id
id +	id T′ E′ $	id * id $	casa com id
id + id	T′ E′ $	* id $	T′→ * F T′
id + id	* F T′ E′ $	* id $	casa com *
id + id *	F T′ E′ $	id $	F → id
id + id * id	id T′ E′ $	id $	casa com id
id + id * id	T′ E′ $	$	T′→ λ
id + id * id	E′ $	$	E′→ λ
id + id * id	$	$	

Analisador recursivo descendente

Para cada não terminal da gramática, uma função é construída. O código da função construída para um não terminal *A* tem um formato como o descrito a seguir:

```
Token lookahead;
...
void A() {
  p = M[A,lookahead];
  if ("p não estiver definido")
    "erro";
  else {
    suponha p uma produção com formato A → X1X2...Xk;
    for (i = 1; i <= k; ++i) {
      if ("Xi é um não terminal")
        Xi(); // chama função Xi
      else    // Xi é um terminal
        if ("Xi igual a lookahead")
          lookahead = "próximo símbolo da entrada";
        else
          "erro";
    }
  }
}
```

O código acima pode ser parcialmente avaliado estaticamente, gerando um código especializado para cada procedimento, uma vez que a tabela M pode ser calculada usando apenas a própria gramática.

Exemplo:

```
void Elinha() {
    if (lookahead == '+') {  // usa produção  E'→ + T E'
        lookahead = "próximo símbolo da entrada";
        T();       // chama o procedimento associado a T
        Elinha();  // e em seguida Elinha, recursivamente
    }
    else if (lookahead == ')' || lookahead == '$') {
        // usa produção  E'→ λ
        // ... nenhuma ação
    }
    else erro("símbolo + , ) ou $ esperados");
}
```