Compiladores

Capítulo 3: Análise Sintática

3.1 - Introdução

O problema da análise sintática para uma dada gramática livre de contexto, consiste em descobrir, para uma cadeia x qualquer, se a cadeia pertence ou não pertence à linguagem da gramática; no caso afirmativo, queremos adicionalmente descobrir a maneira (ou uma das maneiras) pela qual a cadeia pode ser derivada seguindo as regras da gramática.

Esta introdução pretende apenas apresentar a notação usada para representar gramáticas e linguagens livres de contexto, não sendo sua intenção cobrir esse assunto, exceto no que se refere a técnicas de análise sintática para uso em compiladores. Uma boa referência inicial sobre linguagens formais em geral e sobre linguagens e gramáticas livres de contexto em particular é [HopUll79][1].

Um *alfabeto* é um conjunto finito e não vazio, cujos elementos são chamados *símbolos* do alfabeto. Em geral alfabetos são representados por letras gregas maiúsculas (Σ, Δ, Γ, N, ...). Uma cadeia de símbolos no alfabeto Σ é uma seqüência de zero ou mais símbolos de Σ. A *cadeia vazia* ε é a cadeia de zero símbolos, e (por não ter nenhum símbolo) independe do alfabeto usado para defini-la. Uma linguagem no alfabeto Σ é um conjunto de cadeias no alfabeto Σ. Uma linguagem pode ser vazia ou não vazia, finita ou infinita.

Uma gramática G é uma construção matemática usada para definir uma linguagem em um alfabeto Σ, e é definida através de quatro componentes: $G = < \Sigma, N, P, S >$. Temos:

- um alfabeto Σ de *símbolos terminais*, ou simplesmente *terminais*: os símbolos que compõem as cadeias da linguagem.
- um alfabeto N de *símbolos não terminais*, ou simplesmente *nãoterminais*. Os nãoterminais são símbolos auxiliares, que não podem fazer parte das cadeias da linguagem definida pela gramática, que são compostas apenas de terminais.
- um conjunto P de *regras de re-escrita* (ou *regras de produção*) da forma $\alpha \rightarrow \beta$, que indicam que a cadeia α pode ser substituída, onde ocorrer, pela cadeia β. No caso mais geral, as cadeias α e β podem ser compostas de terminais e nãoterminais, em qualquer número, exigindo-se apenas que α contenha pelo menos um nãoterminal.
- um nãoterminal especial S, o *símbolo inicial*.

A linguagem de uma gramática é composta exatamente pelas cadeias que podem ser geradas a partir do símbolo inicial S, usando-se as regras de re-escrita em vários passos até que seja obtida uma cadeia que só tem terminais. Este processo de aplicação das regras de re-escrita é conhecido como derivação. Em um passo de derivação a regra

[1]John E. Hopcroft, Jeffrey D. Ullman, **Introduction to Automata Theory, Languages and Computation**, Addison Wesley, 1979.

$\alpha \rightarrow \beta$ é aplicada a uma cadeia $\gamma\alpha\delta$ tendo como resultado a cadeia $\gamma\beta\delta$. Escrevemos $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$, para indicar este fato. A notação $\gamma \Rightarrow^* \delta$ indica que δ pode ser obtido a partir de γ em zero ou mais passos de derivação.

Formalmente, a linguagem da gramática G é o conjunto

$$L(G) = \{ x \in \Sigma^* \mid S \Rightarrow^* x \},$$

onde Σ^* representa o conjunto das cadeias em Σ, ou seja todas as cadeias compostas de zero ou mais símbolos retirados do alfabeto Σ .

Só nos interessam aqui as gramáticas *livres de contexto*, que constituem um caso particular das gramáticas em geral. Numa gramática livre de contexto as regras de re-escrita têm a forma $A \rightarrow \beta$ com apenas um símbolo A, sempre um nãoterminal, do lado esquerdo.

O nome *livre de contexto* faz referência ao fato de que, *onde quer que apareça*, A pode ser substituído por β, independentemente do *contexto* à sua volta. Por oposição, $\gamma A \delta \rightarrow \gamma\beta\delta$, que permite substituir A por β apenas no contexto "γ antes, δ depois" é uma regra *dependente do contexto* (ou *sensível ao contexto*). A definição de gramática sensível ao contexto permite regras $\alpha \rightarrow \beta$ quaisquer, com a única restrição de que o comprimento de β seja maior ou igual ao comprimento de α.

Na prática, não é necessário indicar explicitamente os quatro componentes de uma gramática livre de contexto, bastando apresentar as regras, seguindo as seguintes convenções:

- os símbolos que aparecem à esquerda nas regras são os nãoterminais, em geral representados por letras latinas maiúsculas (S, A, B, ...).
- os demais símbolos são os terminais, em geral representados por letras latinas minúsculas ou outros símbolos (a, b, c, ..., 0, 1, +, #, ...).
- o símbolo inicial é o que aparece no lado esquerdo da primeira regra.
- várias regras ($A \rightarrow \beta_1$, $A \rightarrow \beta_2$, ..., $A \rightarrow \beta_n$), com um mesmo nãoterminal A do lado esquerdo podem ser reunidas: $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \beta_n$.
- cadeias com terminais e nãoterminais são representadas por letras gregas minúsculas (α, β, γ, ...).
- cadeias só de terminais são representadas por letras latinas (x, y, z, ...).
- um símbolo qualquer, que pode ser terminal ou nãoterminal, é representado por uma letra latina maiúscula (X, Y, ...).

Exemplo 1: Uma das gramáticas mais freqüentemente usadas como exemplo em cursos de linguagens e de compiladores é a gramática G_0, dada por suas regras:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid a \end{array}$$

Pelas convenções vistas, temos para G_0 :

símbolos terminais:	{ +, *, (,), a }
símbolos não terminais:	{ E, T, F }
símbolo inicial:	E
regras:	{ E→E+T, E→T, T→T*F, T→F, F→(E), F→a }

Por exemplo, a cadeia a+a*a pertence à linguagem da gramática, por causa da derivação

(1) E ⇒ E+T ⇒ T+T ⇒ a+T ⇒ a+T*F ⇒ a+F*F ⇒ a+a*F ⇒ a+a*a

Note que outras derivações são possíveis para a mesma cadeia. Por exemplo, temos

(2) E ⇒ E+T ⇒ E+T*F ⇒ E+T*a ⇒ E+F*a ⇒ E+a*a ⇒ T+a*a ⇒ F+a*a ⇒ a+a*a

(3) E ⇒ E+T ⇒ E+T*F ⇒ E+F*F ⇒ E+a*F ⇒ T+a*F ⇒ T+a*a ⇒ F+a*a ⇒ a+a*a

As derivações *(1), (2)* e *(3)* acima são equivalentes, no sentido de que ambas geram a cadeia da mesma maneira, *aplicando as mesmas regras aos mesmos símbolos*, e se diferenciam apenas pela ordem em que as regras são aplicadas. A maneira de verificar isso é usar uma *árvore de derivação*, que para este caso seria a descrita na Fig. 1.

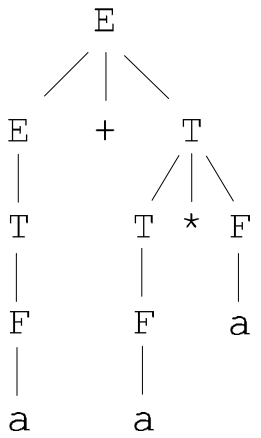


Fig. 1 - Árvore de derivação para a+a*a

□

Num certo sentido, a árvore de derivação caracteriza as gramáticas livres de contexto: a expansão correspondente a cada sub-árvore pode ser feita de forma absolutamente independente das demais sub-árvores. Por essa razão, consideramos equivalentes todas as derivações que correspondem à mesma árvore de derivação. Dois tipos de derivação são especialmente interessantes:

- a *derivação esquerda* (*leftmost derivation*), em que uma regra é sempre aplicada ao primeiro nãoterminal da cadeia, o que fica *mais à esquerda*;
- a *derivação direita (rightmost derivation*), em que uma regra é sempre aplicada ao último nãoterminal da cadeia, o que fica *mais à direita*.

Para especificar uma derivação de uma cadeia x, podemos, equivalentemente, apresentar uma derivação esquerda de x, uma derivação direita de x, ou uma árvore de derivação de x. A partir de uma dessas três descrições, é possível sempre gerar as outras duas. Por exemplo, dada uma árvore de derivação de x, basta percorrer a árvore em pré-ordem (primeiro a raiz, depois as sub-árvores em ordem, da esquerda para a direita), e aplicar as regras encontradas sempre ao primeiro nãoterminal, para construir uma derivação esquerda. Para a derivação direita a árvore deve ser percorrida em uma ordem semelhante: primeiro a raiz, depois as sub-árvores em ordem, da direita para a esquerda.

Exemplo 1 (continuação): A derivação *(1)* é uma derivação esquerda, e a derivação *(2)* é uma derivação direita. A derivação *(3)* nem é uma derivação esquerda, nem é uma derivação direita. Dada uma cadeia x, todas as derivações possíveis de x correspondem exatamente à mesma árvore de derivação. Veremos abaixo que isto quer dizer que a gramática G_0 não é ambígua.

□

Uma gramática é *ambígua* se, para alguma cadeia x, existem duas ou mais árvores de derivação. Podemos mostrar que uma gramática é ambígua mostrando uma cadeia x e duas árvores de derivação distintas de x (ou duas derivações esquerdas distintas, ou duas derivações direitas distintas).

Por outro lado, para mostrar que uma gramática *não* é ambígua, é necessário mostrar que cada cadeia da linguagem tem exatamente uma árvore de derivação. Isso costuma ser feito pela apresentação de um algoritmo para construção (dada uma cadeia qualquer x da linguagem) da única árvore de derivação de x, de uma forma que torne claro que a árvore assim construída é a única árvore de derivação possível para x.

Exemplo 2: Considere a gramática

$$S \rightarrow 0\ S\ 1 \mid \varepsilon$$

A linguagem dessa gramática é formada pelas cadeias da forma 0^n1^n, onde n≥0. Vamos mostrar que essa gramática não é ambígua, mostrando, para cada valor de n, como construir a única derivação de 0^n1^n. (No caso, essa derivação é ao mesmo tempo, uma derivação esquerda e uma derivação direita, porque no máximo há um nãoterminal para ser expandido.)

Para derivar 0^n1^n,

"use n vezes a regra $S \rightarrow 0S1$, e uma vez a regra $S \rightarrow \varepsilon$."

Essa derivação é a única:

- cada aplicação da regra $S \rightarrow 0S1$ introduz um 0 e um 1; as n aplicações introduzem n pares 0, 1.
- a aplicação da regra $S \rightarrow \varepsilon$ é necessária para obter uma cadeia sem nãoterminais.

Por exemplo, para n=3, temos

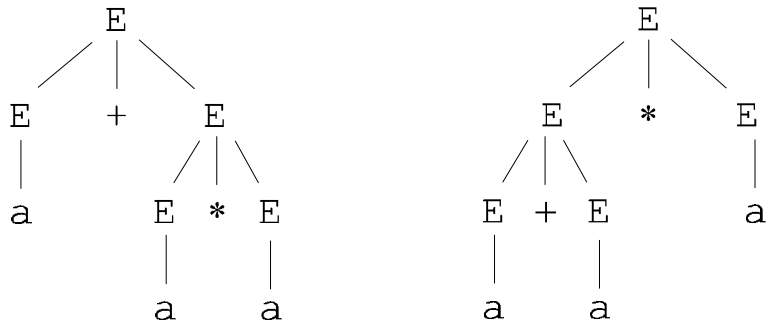
$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$$

□

Exemplo 3: A gramática

$E \rightarrow E+E \mid E*E \mid (E) \mid a$

é ambígua. Por exemplo, a cadeia a+a*a pode ser derivada de duas maneiras diferentes, de acordo com as duas árvores mostradas a seguir:



□

Exercício 1: Mostre que a gramática

$S \rightarrow$ if E then S | if E then S else S | s
$E \rightarrow e$

é ambígua.

□

3.2 - Análise sintática descendente e ascendente

Os métodos de análise sintática podem ser classificados segundo a maneira pela qual a árvore de derivação da cadeia analisada x é construída:

- nos métodos *descendentes*, a árvore de derivação correspondente a x é construída de *cima para baixo*, ou seja, da raiz (o símbolo inicial S) para as folhas, onde se encontra x.
- nos métodos *ascendentes*, a árvore de derivação correspondente a x é construída de *baixo para cima*, ou seja, das folhas, onde se encontra x, para a raiz, onde se encontra o símbolo inicial S.

Nos métodos descendentes (*top-down*), temos de decidir qual a regra $A \rightarrow \beta$ a ser aplicada a um nó rotulado por um nãoterminal A. A *expansão* de A é feita criando nós filhos rotulados com os símbolos de β. Nos métodos ascendentes (*bottom-up*), temos de decidir quando a regra $A \rightarrow \beta$ deve ser aplicada, e devemos encontrar nós vizinhos rotulados com os símbolos de β. A *redução* pela regra $A \rightarrow \beta$ consiste em acrescentar à árvore um nó A, cujos filhos são os nós correspondentes aos símbolos de β.

Métodos descendentes e ascendentes constróem a árvore da esquerda para a direita. A razão para isso é que a escolha das regras deve se basear na cadeia a ser gerada, que é lida da esquerda para a direita. (Seria muito estranho um compilador que começasse a partir do fim do programa fonte!)

Exemplo 4: Considere a gramática

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → a

e a cadeia x = a+a*a, como no exemplo 1. Usando-se um método descendente, a árvore de derivação de x é construída na seqüência especificada. (Figura 2)

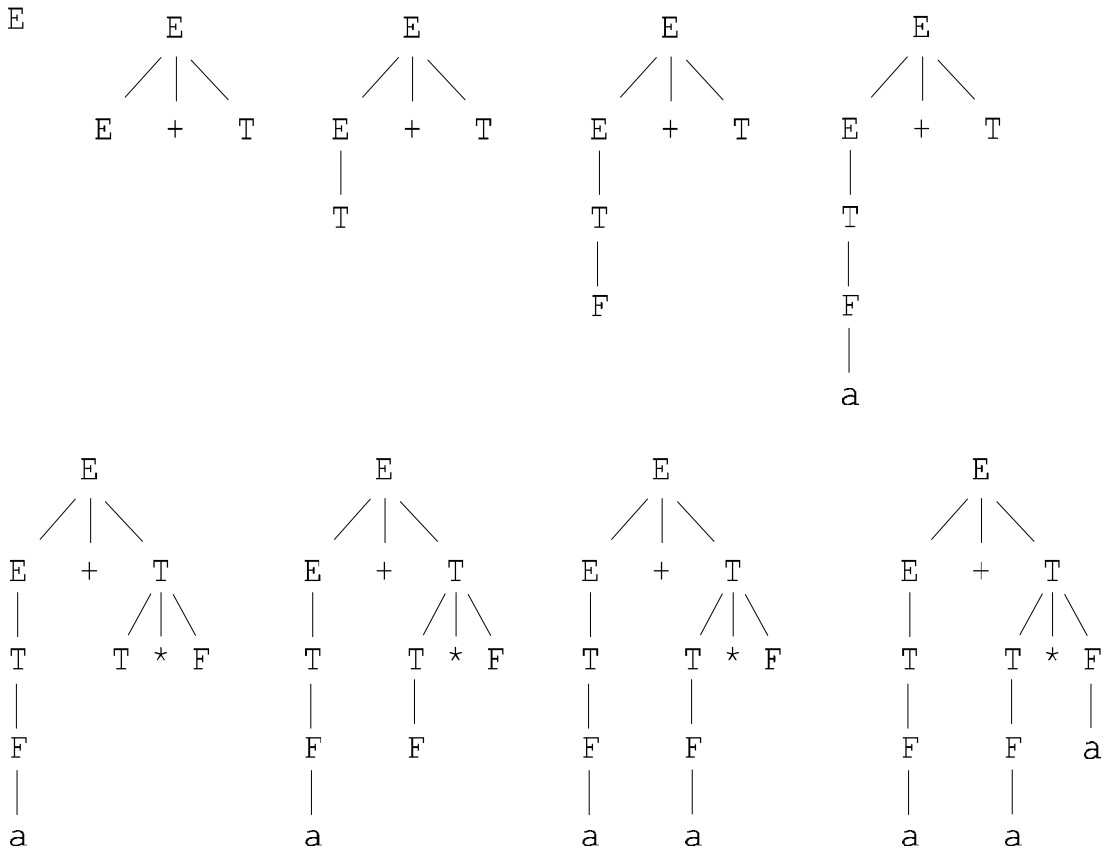


Figura 2 - análise descendente

Note que as regras são consideradas na ordem 1 2 4 6 3 4 6 6, a mesma ordem em que as regras são usadas na derivação esquerda

E ⇒ E+T ⇒ T+T ⇒ a+T ⇒ a+T*F ⇒ a+F*F ⇒ a+a*F
⇒ a+a*a

Usando-se um método de análise ascendente, por outro lado, as regras são identificadas na ordem 6 4 2 6 4 6 3 1, e os passos de construção da árvore podem ser vistos na Figura 3.

Neste caso, a ordem das regras corresponde à derivação direita, *invertida*:

a+a*a ⇐ F+a*a ⇐ T+a*a ⇐ E+a*a ⇐ E+F*a ⇐ E+T*a

$$\Leftarrow E+T*F \Leftarrow E+T \Leftarrow E$$

ou seja,

$$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*a \Rightarrow E+F*a \Rightarrow E+a*a \Rightarrow T+a*a \Rightarrow F+a*a \Rightarrow a+a*a$$

□

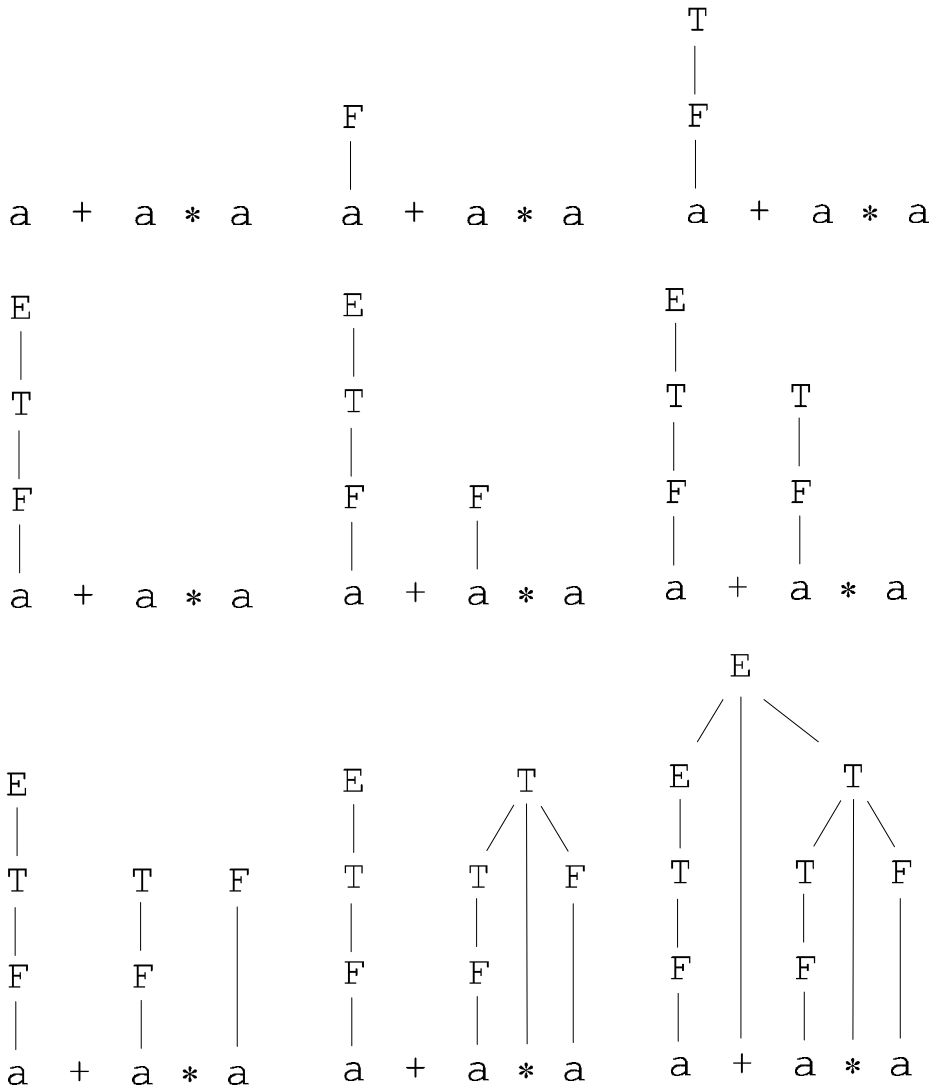


Figura 3 - análise ascendente

Embora a árvore de derivação seja usada para descrever os métodos de análise, na prática ela nunca é efetivamente construída. Às vezes, se necessário, construímos "árvores sintáticas", que guardam alguma semelhança com a árvore de derivação, mas ocupam um espaço de memória significativamente menor. A única estrutura de dados necessária para o processo de análise é uma pilha, que guarda informação sobre os nós da árvore de derivação relevantes, em cada fase do processo. No caso da análise descendente, os nós relevantes são aqueles ainda não expandidos; no caso da análise ascendente, são as raízes das árvores que ainda não foram reunidas em árvores maiores.

Vamos examinar primeiro o caso da análise descendente. A representação do processo será feita através de configurações (α, y), onde α e y representam respectivamente, o conteúdo da pilha e o *resto* da entrada ainda não analisada. Por

convenção vamos supor que *o topo da pilha fica à esquerda*, isto é, que o primeiro símbolo de α é o símbolo do topo da pilha.

Existem duas formas de transição de uma configuração para outra:

1. *expansão de um nãoterminal pela regra* $A \rightarrow \beta$*:* permite passar da configuração $(A\alpha, y)$ para a configuração $(\beta\alpha, y)$.
2. *verificação de um terminal* a*:* permite passar da configuração $(a\alpha, ay)$ para a configuração (α, y).

O segundo tipo de transição serve para retirar terminais do topo da pilha e expor no topo da pilha o próximo nãoterminal a ser expandido.

A configuração inicial para a entrada x é (S, x); o processo termina na configuração $(\varepsilon, \varepsilon)$, com a pilha vazia, e a entrada toda considerada.

Exemplo 4 (continuação): Mostramos abaixo as configurações sucessivas de um analisador descendente, para a cadeia x. Para acompanhamento, a terceira coluna apresenta os passos correspondentes da derivação esquerda de x. Note que apenas os passos de expansão têm correspondente na derivação.

pilha	(resto da) entrada	derivação esquerda
E	a+a∗a	E
E+T	a+a∗a	⇒ E+T
T+T	a+a∗a	⇒ T+T
F+T	a+a∗a	⇒ F+T
a+T	a+a∗a	⇒ a+T
+T	+a∗a	
T	a∗a	
T∗F	a∗a	⇒ a+T∗F
F∗F	a∗a	⇒ a+F∗F
a∗F	a∗a	⇒ a+a∗F
∗F	∗a	
F	a	
a	a	⇒ a+a∗a
ε	ε	

□

Por enquanto, nada foi dito sobre a forma de escolha da regra a ser aplicada. Isso será objeto da seção 3.4, onde falaremos dos analisadores LL(1).

Para a análise ascendente, as configurações também serão da forma (α, y), onde α é o conteúdo da pilha e y é *o resto* da entrada. Entretanto, a convenção sobre o topo da pilha é invertida: o topo fica à direita, ou seja, o último símbolo de α é o símbolo do topo. As mudanças de configuração são

1. *redução pela regra* $A \rightarrow \beta$*:* permite passar da configuração $(\alpha\beta, y)$ para a configuração $(\alpha A, y)$.
2. *empilhamento, ou deslocamento de um terminal* a*:* permite passar da configuração (α, ay) para a configuração $(\alpha a, y)$.

Note que o segundo tipo de transição permite trazer os terminais para a pilha, para que possam participar das reduções, que sempre ocorrem no topo da pilha. A configuração inicial para a entrada x é (ε, x), com a pilha vazia. Ao final, temos a configuração (S, ε), que indica que toda a entrada x foi lida e reduzida para S.

Exemplo 4 (continuação): Mostramos abaixo as configurações sucessivas de um analisador ascendente, para a cadeia x. Para acompanhamento, a terceira coluna apresenta os passos correspondentes da derivação direita de x. Note que apenas os passos de redução têm correspondente na derivação, e que a derivação se apresenta invertida, partindo de x para o símbolo inicial E.

Pilha	(resto da) entrada	derivação direita (invertida)
	a+a∗a	a+a∗a
A	+a∗a	
F	+a∗a	⇐ F+a∗a
T	+a∗a	⇐ T+a∗a
E	+a∗a	⇐ E+a∗a
E+	a∗a	
E+a	∗a	
E+F	∗a	⇐ E+F∗a
E+T	∗a	⇐ E+T∗a
E+T∗	a	
E+T∗a	ε	
E+T∗F	ε	⇐ E+T∗F
E+T	ε	⇐ E+T
E	ε	⇐ E

□

Também neste caso da análise ascendente, nada foi dito sobre a regra a ser escolhida, ou sobre quando se deve escolher a ação de empilhamento. Veremos como essas decisões são feitas nas seções 3.5 e seguintes sobre analisadores LR(1) e suas variantes.

3.3 - Coletando informação sobre gramáticas

Esta seção mostra como obter três tipos de informação sobre símbolos não terminais em gramáticas livres de contexto. Estas informações serão usadas nas próximas seções para decidir quais as regras a serem usadas durante o processo da análise sintática. Especificamente, queremos saber, para um nãoterminal A:

1. se A gera ou não a cadeia vazia ε.
2. quais são os símbolos terminais iniciadores das cadeias geradas a partir de A: se $A \Rightarrow^* \alpha$, que terminais podem aparecer como primeiro símbolo de α.
3. quais são os símbolos terminais seguidores de A: ou seja, se $S \Rightarrow^* \alpha A \beta$, que terminais podem aparecer como primeiro símbolo de β.

Os algoritmos que vamos ver nesta seção só se aplicam, de uma forma geral, a gramáticas que não têm símbolos e/ou regras inúteis, isto é, todos os símbolos e todas as

regras servem para gerar alguma cadeia da linguagem. Na prática, símbolos e regras inúteis ocorrem por engano ou em gramáticas ainda incompletas.

Geração da cadeia vazia. Vamos começar apresentando um algoritmo que determina, para todos os nãoterminais A de uma gramática G, se A gera ou não a cadeia vazia.

Algoritmo 1: Identificação dos nãoterminais que derivam a cadeia vazia ε.

O algoritmo trabalha com uma lista L de regras, que vai sendo alterada pela execução. Se a gramática não tem regras e/ou símbolos inúteis, o algoritmo termina com todos os nãoterminais marcados `sim` (nãoterminal gera ε), ou `não` (nãoterminal não gera ε).

0. Inicialmente a lista L contém todas as regras de G, exceto aquelas que contém um símbolo terminal. (Regras cujo lado direito têm um terminal não servem para derivar ε.)
1. Se existe um nãoterminal A sem regras (não interessa se A originalmente não tinha regras, ou se todas as regras foram retiradas de L) marque A com `não`.
2. Se um nãoterminal A tem uma regra A→ε, retire de L todas as regras com A do lado esquerdo, e retire todas as ocorrências de A do lado direito das regras de L. Marque A com `sim`. (Note que uma regra B→C se transforma em B→ε, se a ocorrência de C do lado direito for retirada.)
3. Se uma regra tem do lado direito um nãoterminal marcado `não`, retire a regra. (Regras cujo lado direito têm um nãoterminal que não deriva ε não servem para derivar ε.)
4. Repita os passos 1, 2, 3 até que nenhuma nova informação seja adicionada.

□

Exemplo 5: Seja a gramática

```
1.   P → A B C D          5.   B → B b
2.   A → ε                6.   C → c
3.   A → a A              7.   C → A B
4.   B → ε                8.   D → d
```

Vamos executar o algoritmo para verificar se `P`, `A`, `B`, `C`, `D` geram a cadeia vazia ε. Inicialmente, as regras 3, 5, 6 e 8 não são incluídas em L por causa dos terminais do lado direito. L contém inicialmente

```
1.   P → A B C D
2.   A → ε
4.   B → ε
7.   C → A B
```

Pelo passo 1, `D`, que não tem regras, é marcado `não`. Como `A` tem uma regra com lado direito vazio, pode ser marcado `sim`, pelo passo 2, que também retira a regra 2. As ocorrências de `A` nas regras 1 e 7 podem ser retiradas, e portanto L contem

```
1.   P → B C D
4.   B → ε
7.   C → B
```

Como B tem uma regra com lado direito vazio, pode ser marcado sim, pelo passo 2, que também retira a regra 4. As ocorrências de B nas regras 1 e 7 podem ser retiradas, e portanto L contem

1. P → C D
7. C → ε

Como C tem uma regra com lado direito vazio, pode ser marcado sim, pelo passo 2, que também retira a regra 7. A ocorrência de C na regra 1 pode ser retirada, e portanto L contem

1. P → D

Pelo passo 3, como D está marcado não, a regra 1 pode ser retirada, deixando a lista L vazia. Como P não tem regras, P pode ser marcado não. Ao final, portanto, temos

sim: A, B, C
não: P, D

Note que C não deriva ε diretamente, mas sim em três passos, por exemplo, através da derivação

C ⇒ A B ⇒ B ⇒ ε.

□

Cálculo dos iniciadores. Formalmente, podemos definir os símbolos iniciadores de um nãoterminal A através de um conjunto First(A)

First(A) = { a | a é um terminal e $A \Rightarrow^* a\alpha$, para alguma cadeia α qualquer }.

O algoritmo a seguir calcula os conjuntos First(A) para todos os nãoterminais A de uma gramática G. O algoritmo se baseia nos seguintes pontos:

- Se há uma regra $A \rightarrow a\alpha$, então $a \in$ First(A). A derivação correspondente é $A \Rightarrow a\alpha$.
- Se há uma regra $A \rightarrow B_1...B_m a\alpha$, e para todo i=1, ..., m, $B_i \Rightarrow^* \varepsilon$, então também temos $a \in$ First(A). Neste caso, a não é o primeiro símbolo, mas passa a ser quando todos os B_i "desaparecerem", isto é, forem substituídos por ε. A derivação correspondente é $A \Rightarrow B_1...B_m a\alpha \Rightarrow^* a\alpha$.
- Se há uma regra $A \rightarrow B\alpha$, e se $a \in$ First(B), temos também $a \in$ First(A). Se $a \in$ First(B), temos $B \Rightarrow^* a\beta$, e a derivação correspondente é $A \Rightarrow B\alpha \Rightarrow^* a\beta\alpha$.
- Se há uma regra $A \rightarrow B_1...B_m B\alpha$, e para todo i=1, ..., m, $B_i \Rightarrow^* \varepsilon$, então se $a \in$ First(B), temos também $a \in$ First(A), de forma semelhante, já que os B_i "desaparecem". Se tivermos $B \Rightarrow^* a\beta$, a derivação correspondente é $A \Rightarrow B_1...B_m B\alpha \Rightarrow B\alpha \Rightarrow^* a\beta\alpha$.

Algoritmo 2: Cálculo de First(A), para todos os nãoterminais A de uma gramática G.

0. Inicialmente, para todos os nãoterminais A da gramática G, todos os conjuntos First(A) estão vazios.
1. Para cada regra $A \rightarrow B_1...B_m a\alpha$, tal que para todo i=1, ..., m, $B_i \Rightarrow^* \varepsilon$, acrescente a a First(A).

2. Para cada regra $A \rightarrow B_1 ... B_m B\alpha$, tal que, para todo i=1, ..., m, $B_i \Rightarrow^* \varepsilon$, acrescente First(B) a First(A).
3. Repita o passo 2 enquanto houver alteração no valor de algum dos conjuntos First.

Exemplo 6: Considere a gramática do Exemplo 1.

```
1.   E → E + T      2.   E → T
3.   T → T * F      4.   T → F
5.   F → ( E )      6.   F → a
```

Vamos calcular aplicar o algoritmo acima para calcular os conjuntos de iniciadores de E, T e F. Note que nenhum dos nãoterminais deriva ε.

Passo 0. First(E) = First(T) = First(F) = ∅.

Passo1. Pela regra 5, acrescentamos (a First(F). Semelhantemente, pela regra 6, acrescentamos a ao mesmo First(F). Temos:

First(E) = First(T) = ∅.
First(F) = { (, a }.

Passo 2. Pela regra 4, acrescentamos First(F) a First(T). Temos

First(E) = ∅.
First(T) = First(F) = { (, a }.

Pela regra 2, acrescentamos First(T) a First(E). Temos então

First(E) = First(T) = First(F) = { (, a }.

Esta é a resposta final. (Não foram indicadas as aplicações do *Passo 2* que não acrescentam nenhum elemento ao conjunto correspondente.)

□

Exemplo 7: Considere a gramática do Exemplo 5.

```
1.   P → A B C D          5.   B → B b
2.   A → ε                6.   C → c
3.   A → a A              7.   C → A B
4.   B → ε                8.   D → d
```

Já sabemos, do Exemplo 5, que A, B e C geram ε, o que não acontece com P e D. Vamos agora calcular os conjuntos de iniciadores. Inicialmente,

First(P) = First(A) = First(B) = First(C) = First(D) = ∅.

Vamos executar primeiro o Passo 1. Pela regra 3, acrescentamos a a First(A); pela regra 5, notando que $B \Rightarrow^* \varepsilon$, acrescentamos b a First(B); pela regra 6, acrescentamos c a First(C), e, finalmente, pela regra 8, acrescentamos d a First(D). Temos agora

First(P) = ∅.
First(A) = { a }
First(B) = { b }

First(C) = { c }
First(D) = { d }

Vamos agora executar o passo 2. Pela regra 1, acrescentamos First(A) a First(P); pela mesma regra, como $A \Rightarrow^* \varepsilon$, acrescentamos First(B) a First(P); pela mesma regra, como $A \Rightarrow^* \varepsilon$ e $B \Rightarrow^* \varepsilon$, acrescentamos First(C) a First(P); pela mesma regra, como $A \Rightarrow^* \varepsilon$, $B \Rightarrow^* \varepsilon$, e $C \Rightarrow^* \varepsilon$, acrescentamos First(D) a First(P). Pela regra 7, acrescentamos First(A) a First(C); pela mesma regra, já que $A \Rightarrow^* \varepsilon$, acrescentamos First(B) a First(C). Temos agora

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

Este é o resultado final, porque a aplicação adicional do passo 2 às regras não acrescenta nenhum elemento aos conjuntos.

□

Cálculo dos seguidores. Um ponto a considerar aqui é o de que um nãoterminal pode aparecer no fim da cadeia derivada, e portanto, não ter nenhum símbolo seguidor nessa situação. Para que este caso seja tratado juntamente com os demais, vamos introduzir um símbolo novo, $, que será tratado como se fosse um símbolo terminal, e que indica o fim da cadeia. Do ponto de vista de implementação, $ tem uma interpretação simples: é a marca de fim-de-arquivo do arquivo fonte, e pode ser devolvido pelo analisador léxico da mesma forma que os outros tokens. Para levar esse símbolo especial em consideração, vamos incluí-lo na definição do Follow (o conjunto de seguidores) após o símbolo inicial S. Consequentemente, $ será encontrado como seguidor de qualquer símbolo nãoterminal que apareça no fim de uma cadeia gerada a partir de S, e, em particular, sempre teremos $ como seguidor de S. Formalmente, podemos definir os símbolos seguidores de um nãoterminal A através de um conjunto Follow(A)

Follow(A) = { a | a é um terminal e $S\$ \Rightarrow^* \alpha A a \beta$, para cadeias α, β quaisquer }.

O algoritmo a seguir calcula os conjuntos Follow(A) para todos os nãoterminais A de uma gramática G. No que se segue, $\gamma = B_1 \ldots B_m$ é uma cadeia de nãoterminais que derivam ε, e, portanto, $\gamma \Rightarrow^* \varepsilon$. Suporemos também que $S\$ \Rightarrow^* \delta A \varphi$. O algoritmo se baseia nos seguintes pontos:

- Como $S\$ \Rightarrow^* S\$$, $\$ \in$ Follow(S).
- Se há uma regra $A \rightarrow \alpha B \gamma a \beta$, então $a \in$ Follow(B). A derivação correspondente é $S\$ \Rightarrow^* \delta A \varphi \Rightarrow^* \delta \alpha B \gamma a \beta \varphi \Rightarrow^* \delta \alpha B a \beta \varphi$
- Se há uma regra $A \rightarrow \alpha B \gamma C \beta$, então se $a \in$ First(C), temos também $a \in$ Follow(B). Se $a \in$ First(C), temos $C \Rightarrow^* a\mu$. A derivação correspondente é então $S\$ \Rightarrow^* \delta A \varphi \Rightarrow^* \delta \alpha B \gamma C \beta \varphi \Rightarrow^* \delta \alpha B C \beta \varphi \Rightarrow^* \delta \alpha B a \mu \beta \varphi$
- Se há uma regra $A \rightarrow \alpha B \gamma$, então se $a \in$ Follow(A), temos também $a \in$ Follow(B). Se $a \in$ Follow(A), temos $S\$ \Rightarrow^* \delta A a \varphi$. A derivação correspondente é então $S\$ \Rightarrow^* \delta A a \varphi \Rightarrow^* \delta \alpha B \gamma a \varphi \Rightarrow^* \delta \alpha B a \varphi$.

Algoritmo 3: Cálculo de Follow(A), para todos os nãoterminais A de uma gramática G.

0. Inicialmente, para todos os nãoterminais A da gramática G, todos os conjuntos Follow(A) estão vazios, excetuando-se Follow(S)= { $ }.
1. Se há uma regra $A \rightarrow \alpha B \gamma a \beta$, e $\gamma = B_1 \ldots B_m \Rightarrow^* \varepsilon$, então acrescente a a Follow(B).
2. Se há uma regra $A \rightarrow \alpha B \gamma C \beta$, e $\gamma = B_1 \ldots B_m \Rightarrow^* \varepsilon$, então acrescente First(C) a Follow(B).
3. Se há uma regra $A \rightarrow \alpha B \gamma$, e $\gamma = B_1 \ldots B_m \Rightarrow^* \varepsilon$, então acrescente Follow(A) a Follow(B).
4. Repita o passo 3 enquanto houver modificação em algum dos conjuntos.

□

Exemplo 8: Considere a gramática dos Exemplos 1 e 6.

```
1.   E → E + T      2.   E → T
3.   T → T * F      4.   T → F
5.   F → ( E )      6.   F → a
```

Vamos aplicar o algoritmo de cálculo dos conjuntos Follow. Sabemos que nenhum dos nãoterminais deriva ε, e que

First(E) = First(T) = First(F) = { (, a }.

Para aplicar o algoritmo, temos inicialmente

Follow(E) = { $ }
Follow(T) = Follow(F) = ∅.

Passo 1. Pela regra 1, acrescentamos + a Follow(E); pela regra 3, acrescentamos * a Follow(T); pela regra 5, acrescentamos) a Follow(E). Ao final, temos:

Follow(E) = { $, +,) }
Follow(T) = { * }
Follow(F) = ∅.

Passo 2. Pela regra 2, Follow(E) deve ser acrescentado a Follow(T). Pela regra 4, Follow(T) deve ser acrescentado a Follow(F).

O resultado final é

Follow(E) = { $, +,) }
Follow(T) = Follow(F) = { $, +,), * }.

□

Exemplo 9: Considere a gramática

```
1.   E → T E'       2.   T → F T'
3.   F → ( E )      4.   F → a
5.   E' → + T E'    6.   E' → ε
7.   T' → * F T'    8.   T' → ε
```

Para este exemplo, vamos verificar quais os nãoterminais que derivam a cadeia vazia, e calcular os conjuntos First e Follow.

Geração da cadeia vazia. Pelas regras 6 e 8, a resposta para E' e T' é `sim`. Pelas regras 3 e 4, que contêm terminais, a resposta para F é `não`. Como F aparece do lado direito da única regra de T, a resposta para T é `não`. Como T aparece do lado direito da única regra de E, a resposta para E também é `não`.

Conjuntos First. Pelas regras 3 e 4, (e a pertencem a First(F). Pela regra 5, + pertence a First(E'). Pela regra 7, ∗ pertence a First(T'). Pela regra 2, First(F) é acrescentado a First(T); pela regra 1, First(T) é acrescentado a First(E). Temos

First(E) = First(T) = First(F) = { (, a }.
First(E') = { + }. First(T') = { ∗ }.

Conjuntos Follow. Inicialmente, $ pertence a Follow(E).

(Passos 1 e 2) Pela regra 1, First(E') deve ser acrescentado a Follow(T); pela regra 2, First(T') deve ser acrescentado a Follow(F); pela regra 3,) deve ser acrescentado a Follow(E). Neste momento temos

Follow(E) = { $,) }. Follow(T) = { + }
Follow(F) = { ∗ } Follow(E') = Follow(T') = ∅.

(Passo 3) Pela regra 1, Follow(E) deve ser acrescentado a Follow(E'), e, como E'⇒*ε, Follow(E) deve também ser acrescentado a Follow(T). Pela regra 2, Follow(T) deve ser acrescentado a Follow(T'), e, como T'⇒*ε, Follow(T) deve também ser acrescentado a Follow(F). Semelhantemente, pelas regras 5 e 7, Follow(E') deve ser acrescentado a Follow(T), e Follow(T') a Follow(F), mas essas operações não trazem nenhum símbolo adicional.

O resultado final é

Follow(E) = { $,) }. Follow(T) = { $,), + }
Follow(F) = { $,), +, ∗ } Follow(E') = { $,) }
Follow(T') = { $,), + }

□

Exercício 2: Para a gramática do Exemplo 9, escreva derivações que justifiquem a presença de cada terminal, em cada conjunto First e em cada conjunto Follow.

□

Um conceito que será necessário na seção seguinte é o conceito de conjunto de símbolos iniciadores de uma cadeia qualquer α, ou seja, First(α). A definição é feita recursivamente:

se α=ε,
  First(α)=First(ε)=∅;
se α é um terminal a,
  First(α)=First(a)= { a };
se α é um nãoterminal,
  First(A) é calculado pelo algoritmo 2;
se α é uma cadeia Aβ, e o primeiro símbolo é um nãoterminal A, que deriva ε,
  First(α)=First(Aβ)=First(A) ∪ First(β)

se α é uma cadeia $A\beta$, e o primeiro símbolo é um nãoterminal A,
que não deriva ε,
First(α)=First($A\beta$)=First(A)
se α é uma cadeia $a\beta$, cujo primeiro símbolo é um terminal a,
First(α)=First($a\beta$)={ a }

Exemplo 10: Considere a gramática do Exemplo 9. Seja calcular First(T'E')T'E'). Temos, pela definição acima:

First(T'E')T'E') = First(T') $\cup$ First(E')T'E') = First(T') $\cup$ First(E') $\cup$ First()T'E')
= First(T') $\cup$ First(E') $\cup$ First()) = { +, *,) }. □

3.4 - Análise sintática LL(1).

Podemos agora apresentar uma resposta à questão da escolha da regra a ser usada durante o processo de análise descendente. A idéia é utilizar duas informações: o nãoterminal A a ser expandido e o primeiro símbolo a do resto da entrada. Uma tabela M com essas duas entradas dá a regra a ser utilizada: M[A, a]. Essa técnica só pode ser usada para uma classe restrita de gramáticas, a classe das gramáticas LL(1). O nome LL(1) indica que

- a cadeia de entrada é examinada da esquerda para a direita (L=*left-to-right*);
- o analisador procura construir uma derivação esquerda (L=*leftmost*);
- exatamente *1* símbolo do resto da entrada é examinado.

Exemplo 11: Considere a gramática dos Exemplos 9 e 10.

```
1.   E → T E'        2.   T → F T'
3.   F → ( E )       4.   F → a
5.   E' → + T E'     6.   E' → ε
7.   T' → * F T'     8.   T' → ε
```

Essa gramática é LL(1), como veremos posteriormente, e a tabela de análise M correspondente é

	(	a	+	*	)	$
E	1	1	-	-	-	-
T	2	2	-	-	-	-
F	3	4	-	-	-	-
E'	-	-	5	-	6	6
T'	-	-	8	7	8	8

Nessa tabela, a entrada M[A, a] correspondente ao nãoterminal A e ao terminal a tem o número da regra que deve ser usada para expansão de A. As entradas indicadas por "-" correspondem a erros, isto é combinações que não podem ocorrer durante a análise de cadeias da linguagem. Para analisar a cadeia a+a*a, teremos as seguintes configurações:

Pilha	Entrada	escolha da regra
E	a+a*a	M[E, a] = 1
TE'	a+a*a	M[T, a] = 2
FT'E'	a+a*a	M[F, a] = 4

Pilha	Entrada	escolha da regra
aT'E'	a+a∗a	-
T'E'	+a∗a	M[T', +] = 8
E'	+a∗a	M[E', +] = 5
+TE'	+a∗a	-
TE'	a∗a	M[T, a] = 2
FT'E'	a∗a	M[F, a] = 4
aT'E'	a∗a	-
T'E'	∗a	M[T', ∗] = 7
∗FT'E'	∗a	-
FT'E'	a	M[F, a] = 4
aT'E'	a	-
T'E'	ε	M[T', $] = 8
E'	ε	M[E', $] = 6
ε	ε	-

Vamos agora mostrar como construir M, a tabela de análise LL(1). No caso mais simples, o símbolo a (o primeiro do resto da entrada) é o primeiro símbolo derivado do nãoterminal a ser expandido A, e faz parte de First(A). Neste caso, a deve pertencer a First(α), onde A→α é uma das alternativas de regra para A. Como ilustração, podemos ver que, no exemplo acima, a regra F→(E) foi usada com o símbolo (.

Outra possibilidade é a de que não seja A o nãoterminal responsável pela geração do símbolo a, mas sim algum outro nãoterminal encontrado depois de A. Neste caso, devemos ter a pertencendo ao Follow(A). Como ilustração, podemos ver que, no exemplo acima, a regra T'→ε foi usada com o símbolo +.

Para construir a tabela M, vamos examinar todas as regras da gramática:

- Para cada regra i: A→α, temos M[A, a]=i, para cada a em First(α).
- Para cada regra i: A→α, se α⇒*ε, temos M[A, a]=i, para cada a em Follow(A).

Se a gramática é LL(1), cada entrada de M receberá no máximo um valor. As entradas de M que não receberem nenhum valor devem ser marcadas como entradas de erro. Caso alguma entrada de M seja definida mais de uma vez, dizemos que houve um conflito, e que a gramática não é LL(1).

Exemplo 12: Considere a gramática do Exemplo 9. Essa gramática é LL(1), e a tabela de análise M pode ser construída como indicado. Temos, para cada regra:

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

7. T'→ +FT'	First(∗FT') = { ∗ }	M[T', ∗] = 7
8. T'→ ε	Follow(T') = { $, +,) }	M[T', $] = M[T', +] = M[T',)] = 8

Portanto, está correta a tabela usada no Exemplo 11.

□

Exemplo 13: Considere a gramática do Exemplo 1. Temos

1. E→E+T	First(E+T)= { (, a }	M[E, (] = M[E, a] = 1
2. E→T	First(T)= { (, a }	M[E, (] = M[E, a] = 2
3. T→T∗F	First(T∗F)= { (, a }	M[T, (] = M[T, a] = 3
4. T→F	First(F)= { (, a }	M[T, (] = M[T, a] = 4
5. F→(E)	First((E))= { (}	M[F, (] = 5
6. F→a	First(a)= { a }	M[F, a] = 6

Consequentemente, a gramática não é LL(1), por causa dos conflitos (múltiplas definições) para M[E, (], M[E, a], M[T, (] e M[T, a].

□

Em alguns casos, como o da gramática do Exemplo 13, é possível concluir que a gramática não é LL(1) por inspeção. As duas características mais óbvias são a recursão à esquerda e a possibilidade de fatoração.

Recursão à esquerda. Se uma gramática permite uma derivação $A \Rightarrow^* A\alpha$, para algum nãoterminal A e para alguma cadeia não vazia α, a gramática é dita recursiva à esquerda. No caso mais simples, que vamos examinar aqui, existe na gramática uma regra $A \rightarrow A\alpha$, responsável diretamente pela derivação mencionada.

Naturalmente, para que A não seja um nãoterminal inútil, deve existir na gramática (pelo menos) uma regra da forma $A \rightarrow \beta$, sem recursão à esquerda. A combinação dessas duas regras faz com que First(A) e, portanto, First(Aα) contenham todos os símbolos de First(β), e isso leva necessariamente a um conflito.

A eliminação da recursão à esquerda pode ser tentada, procurando transformar a gramática em uma gramática LL(1). Basta observar que a combinação

$A \rightarrow A\alpha$
$A \rightarrow \beta$

permite a geração de cadeias da forma $\beta\alpha\alpha...\alpha$, e que essas mesmas cadeias podem ser geradas de outra maneira. Por exemplo,

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

Esta outra forma de geração usa recursão à direita, que não cria problemas.

Possibilidade de fatoração. Outra combinação interessante é a de duas regras que começam pelo mesmos símbolos, isto é, regras como $A \rightarrow \alpha\beta$ e $A \rightarrow \alpha\gamma$, com First($\alpha$)≠∅. Neste caso, existe uma interseção entre First($\alpha\beta$) e First($\alpha\gamma$), e a gramática não pode ser LL(1). Isto acontece porque não é possível decidir, olhando apenas o primeiro símbolo

derivado de α, qual a regra correta. A solução a ser tentada é simples, e envolve a fatoração: a cadeia inicial α é *posta em evidência*. Temos

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

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

Estas duas técnicas (eliminação de recursão à esquerda, fatoração) permitem transformar algumas gramáticas em gramáticas LL(1). Entretanto, devemos observar que algumas gramáticas livres de contexto não têm gramáticas equivalentes LL(1), e nesse caso a aplicação dessas (ou de outras) técnicas não poderá ter sucesso.

Exemplo 14: Considere a gramática do Exemplo 1:

```
E → E + T     |    T
T → T * F     |    F
F → ( E )     |    a
```

Esta gramática apresenta duas situações de recursão à esquerda. Fazendo as substituições indicadas, temos

```
E → T E'
E' → + T E'     |    ε
T → F T'
T' → * F T'     |    ε
F → ( E )       |    a
```

e esta gramática é LL(1), como já foi visto no Exemplo 9 e seguintes.

□

Exemplo 15: Seja transformar a gramática abaixo em uma gramática LL(1) equivalente.

```
L → L ; S |   S
S → if E th L el L fi
   | if E th L fi | s
E → e
```

Temos recursão à esquerda nas regras de L, e possibilidade de fatoração 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
```

Podemos verificar que esta é uma gramática LL(1). A construção da tabela do analisador fica como exercício.

□

Exemplo 16: Considere a gramática

```
S → if E th S el S | if E th S | s
E → e
```

Inicialmente, é aparente que, pela possibilidade de fatoração nas duas primeiras regras, esta gramática não pode ser LL(1). Entretanto, como, além disso, a gramática é ambígua, a fatoração correspondente não tem sucesso, mas nos leva a

```
1.   S → if E th S S'
2.     | s
3.   S' → el S
4.     | ε
5.   E → e
```

A tentativa de construir a tabela LL(1) para esta gramática nos leva a

	if	s	el	$	e
S	1	2	-	-	-
S'			3/4	4	-
E					5

A entrada 3/4 da tabela indica um conflito. Se substituirmos esta entrada por 3, teremos um analisador sintático (quase LL(1)), que funciona para a gramática ambígua a partir da qual foi gerado. Essa *cirurgia* se reflete nos manuais de linguagens de programação pela regra "em caso de ambigüidade, a parte `else` corresponde ao último `if` aberto". Como a *linguagem* da gramática considerada não é LL(1), isto é, não tem uma gramática LL(1), só há duas soluções: a *cirurgia* mencionada e o uso de um `if` "fechado", como no Exemplo 15.

□

Descida recursiva. Uma forma de análise descendente é a técnica de descida recursiva (*recursive descent*), em que cada símbolo nãoterminal se transforma em um procedimento. Uma chamada de um procedimento A tem a finalidade de encontrar a maior cadeia que pode ser derivada do nãoterminal A, a partir do ponto inicial de análise. Se a gramática considerada é LL(1), a construção dos procedimentos é automática. Quando a tabela LL(1) prevê o uso da regra $A \rightarrow X_1 X_2 \ldots X_n$, o procedimento A faz uma série de chamadas correspondentes a X_1, X_2, $\ldots X_n$: se X_i é um nãoterminal, o procedimento X_i é chamado; se X_i é um terminal, fazemos uma chamada `check(`X_i`)` que verifica a presença do terminal X_i, e aciona o analisador léxico, para a obtenção do próximo símbolo. O processo de construção está demonstrado no Exemplo seguinte.

Exemplo 17: Considere a gramática do Exemplo 9, cuja tabela LL(1) foi construída no Exemplo 11. Vamos construir um analisador de descida recursiva para essa gramática, supondo disponível um procedimento `scan` que, a cada chamada, atualiza o próximo símbolo `simb`, do tipo `simbs`. O procedimento `erro` deve ser escrito de forma a tomar as ações apropriadas a cada situação. Algumas chamadas de `check` estão substituídas por chamadas de `scan`, quando não há necessidade de testar o símbolo.

```
type simbs=(abre,a,mais,vezes,fecha,dolar);
```

```
procedure check(s:simbs);
begin
     if simb=s then
          scan
     else
          erro
end;

procedure E;
begin
     case simb of
          abre,a:                    { regra 1. E→TE' }
               begin T; Elinha; end;
          mais,vezes,fecha,dolar:
               erro;
     end;
end;

procedure T;
begin
     case simb of
          abre,a:                    { regra 2. T→FT' }
               begin F; Tlinha; end;
          mais,vezes,fecha,dolar:
               erro;
     end;
end;

procedure F;
begin
     case simb of
          abre:                      { regra 3. F→(E) }
               begin scan; E; check(fecha); end;
          a:                         { regra 4. F→a }
               scan;
          mais,vezes,fecha,dolar:
               erro;
     end;
end;

procedure Elinha;
begin
     case simb of
          mais:                      { regra 5. E'→+TE' }
               begin scan; T; Elinha; end;
          fecha,dolar:               { regra 6. E'→ε }
               ;
          abre,a,vezes:
               erro;
     end;
end;
```

```
procedure Tlinha;
begin
     case simb of
          vezes:                    { regra 7. T'→*FT' }
               begin scan; F; Tlinha; end;
          mais,fecha,dolar:         { regra 8. T'→ε }
               ;
          abre,a:
               erro;
     end;
end;
```

□

Por outro lado, se a gramática não é LL(1), ainda é possível construir um analisador de descida recursiva, se bem que a construção deixa de ser automática. Na prática, este é o caso mais interessante, uma vez que no caso de uma gramática LL(1), o analisador LL(1) é sempre mais eficiente do que o de descida recursiva.

Exemplo 18: Seja construir um analisador de descida recursiva para a gramática do Exemplo 1

```
1.   E → E + T      2.   E → T
3.   T → T * F      4.   T → F
5.   F → ( E )      6.   F → a
```

Como a gramática não é LL(1), a construção deve levar em consideração outras propriedades da gramática. Por exemplo, notamos que o uso das regras 1 e 2 leva a cadeias da forma `T +T +T ... +T`, em que o grupo `+T` é repetido zero ou mais vezes; notamos uma propriedade semelhante para as regras 3 e 4. Usando essas propriedades, obtemos os seguintes procedimentos `E`, `T` e `F`.

```
procedure E;
begin
     T;
     while simb=mais do begin
          scan;
          T;
     end;
end;

procedure T;
begin
     F;
     while simb=vezes do begin
          scan;
          F;
     end;
end;
```

```
procedure F;
begin
     case simb of
          abre: begin scan; E; check(fecha); end;
          a: scan;
          mais,vezes,fecha,dolar: erro;
     end;
end;
```

□

3.5 Análise sintática sLR(1)

O primeiro método de análise sintática ascendente que vamos ver é chamado sLR(1). As letras indicam:

s a variante mais *simples* dos métodos LR(1).
L a cadeia de entrada é lida da esquerda para a direita (*left-to-right*).
R o método constrói uma derivação direita (*rightmost*) invertida.
(1) apenas 1 símbolo do resto da entrada é examinado.

Para simplificar a identificação do término do processo de análise, acrescentamos à gramática uma nova regra inicial S'→S, sendo S o símbolo inicial original, e S' um símbolo novo, que passa a ser o símbolo inicial da *gramática aumentada*. Essa regra recebe o número 0. Assim, uma redução pela regra 0 indica o fim da análise, já que S' nunca aparece à direita nas regras da gramática. A gramática aumentada é usada na construção do analisador sLR(1) da gramática original.

A construção deste analisador se baseia em itens. Um *item* A→α•β indica a possibilidade de que, no ponto atual em que se encontra a análise,

- a regra A→αβ foi usada na derivação da cadeia de entrada;
- os símbolos terminais derivados de α já foram encontrados;
- falta encontrar os símbolos terminais derivados de β.

Assim o ponto (•) indica o progresso da análise. Por exemplo, nos casos extremos, A→•γ indica o início da busca por (uma cadeia derivada de) um γ, e A→γ• indica o fim da busca por um γ, ou seja, o momento em que a redução de γ para A pode ser executada.

Num dado momento, várias possibilidades precisam ser consideradas, e, por essa razão, representamos um *estado* do processo de análise por um *conjunto de itens*. O estado inicial do processo de análise tem um item inicial S'→•S, proveniente da regra inicial, e pode ser entendido como "só falta encontrar (uma cadeia derivada de) um S". De acordo com as regras da gramática, é necessário acrescentar a cada estado as possibilidades correspondentes. Assim, quando um estado contém um item A→α•Bβ, itens correspondentes às regras de B devem ser acrescentados, para dirigir a busca por B. Esses são os itens da forma B→•γ, para todas as regras B→γ de B. Esse processo, repetido enquanto for necessário, é denominado o *fechamento* do estado. Assim, o estado inicial é o fechamento de { S'→•S }.

Exemplo 19: Considere a gramática do Exemplo 1. A gramática aumentada é

```
0.   S' → E
1.   E → E + T
2.   E → T
3.   T → T * F
4.   T → F
5.   F → ( E )
6.   F → a
```

Os itens da gramática aumentada são 20: S'→•E, S'→E•, E→•E+T, E→E•+T, E→E+•T, E→E+T•, ..., F→•a, F→a•.

O estado inicial é o fechamento de { S'→•E }. Como há um ponto antes de E, devemos acrescentar os itens E→•E+T e E→•T. Por causa do ponto antes de T, acrescentamos T→•T*F e T→•F. Por causa do ponto antes de F, acrescentamos F→•(E) e F→•a. O estado inicial (estado 0) é então composto pelos itens

S'→•E, E→•E+T, E→•T, T→•T*F, T→•F, F→•(E), F→•a.

□

À medida que a análise prossegue, o ponto deve caminhar para a direita nos diversos itens do estado. Encontrado um símbolo X, passamos de um item A→α•Xβ para um item A→αX•β. Símbolos terminais são encontrados na entrada; símbolos não terminais são encontrados como produto de reduções. Se estamos em um estado p que tem itens com o ponto antes de um símbolo X, a transição com o símbolo X nos leva a outro estado q que tem um item com o ponto depois da ocorrência de X para cada item no estado p com o ponto antes de X. Os outros itens de q são itens obtidos por seu fechamento. Escrevemos q=δ(p). A *tabela de transições* do analisador representa a *função de transição* δ. Essa tabela é às vezes chamada de *tabela GOTO*.

Para gerar todos os estados do analisador, geramos todos os estados possíveis a partir do estado inicial e de outros estados gerados a partir dele. Cada vez que um estado é obtido, verificamos se já ocorreu anteriormente. O número de estados (conjuntos de itens) é finito, uma vez que o número de itens é finito.

Exemplo 19 (continuação):

A partir do estado inicial 0, temos transições com os símbolos E, T, F, (e a, para os estados novos 1, 2, 3, 4 e 5. Por exemplo, com o nãoterminal E, a transição para o estado 1 envolve os itens S'→•E e E→•E+T. Portanto o estado 1 deve conter os itens S'→E• e E→E•+T. Neste caso, não há nãoterminais a considerar, e nenhum item é acrescentado pelo fechamento. Como outro exemplo, com o terminal (, a transição para o estado 4 envolve apenas o item F→•(E). Portanto, o estado 4 deve conter F→(•E), e os itens provenientes do fechamento do estado: E→•E+T, E→•T, E→•T, T→•T*F, T→•F, F→•(E) e F→•a.

A coleção completa de estados é a seguinte:

0.	S'→•E E→•E+T E→•T T→•T∗F T→•F F→•(E) F→•a	4.	F→(•E) E→•E+T E→•T T→•T∗F T→•F F→•(E) F→•a	7.	T→T∗•F F→•(E) F→•a
1.	S'→E• E→E•+T	5.	F→a•	8.	F→(E•) E→E•+T
2.	E→T• T→T•∗F	6.	E→E+•T T→•T∗F T→•F F→•(E) F→•a	9.	E→E+T• T→T•∗F
3.	T→F•			10.	T→T∗F•
				11.	F→(E)•

As transições entre os estados estão especificadas na tabela abaixo:

	E	T	F	(	a	+	∗	)	$
0	1	2	3	4	5	-	-	-	-
1	-	-	-	-	-	6	-	-	-
2	-	-	-	-	-	-	7	-	-
3	-	-	-	-	-	-	-	-	-
4	8	2	3	4	5	-	-	-	-
5	-	-	-	-	-	-	-	-	-
6	-	9	3	4	5	-	-	-	-
7	-	-	10	4	5	-	-	-	-
8	-	-	-	-	-	6	-	11	-
9	-	-	-	-	-	-	7	-	-
10	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-

Note que a tabela inclui alguns estados a partir dos quais não há transições.

□

O analisador sLR(1) é um analisador ascendente. Em vez de símbolos, entretanto, a pilha do analisador contem os estados correspondentes aos símbolos. Primeiro, observamos que a cada estado q, com exceção do estado inicial, corresponde exatamente um símbolo X, que é o único símbolo que ocorre depois do ponto, nos itens do estado q. Todas as transições para q são feitas com o símbolo X, podendo ocorrer, entretanto, que dois ou mais estados sejam acessíveis pelo mesmo símbolo X. Neste caso, os estados se distinguem por conter informação adicional sobre a posição em que o símbolo X ocorre na cadeia de entrada.

As duas ações possíveis em analisadores ascendentes se aplicam aqui. Um empilhamento (*shift*) pode ocorrer quando existe uma transição com um terminal a partir do estado corrente (o estado do topo da pilha). Quando existe um item completo B→γ•, no estado

corrente, pode ser feita uma redução pela regra B→γ. *Em um analisador sLR(1),* a regra é:

- reduza pela regra B→γ se o símbolo da entrada pertencer ao Follow(B).

Esse primeiro símbolo (do resto) da entrada é conhecido como o símbolo de *lookahead.*

Podemos agora construir uma tabela para o analisador sLR(1) que pode conter as seguintes ações, em função do estado q do topo da pilha e do símbolo s de lookahead:

- *empilhamento* - o empilhamento do estado p (que representa s) deve ser empilhado, e o analisador léxico deve ser acionado para obter outro símbolo da entrada.
- *redução* - se T[q, s] = *reduce* B→γ, os |γ| estados correspondentes a γ devem ser retirados da pilha, e o estado δ(q, B) deve ser empilhado, representando B.
- *aceitação* - se T[q, s] = *reduce* S'→S, o processo se encerra com sucesso.

Nos exemplos, uma ação de empilhamento *shift* q será representada apenas pelo número do estado q, e uma ação de redução *reduce* B→γ, será representada por *r i*, onde *i* é o número da regra B→γ; a ação de parada será indicada como *r0*.

Exemplo 19: (continuação)

A tabela de ações é a seguinte:

	E	T	F	(	a	+	*	)	$
0	1	2	3	4	5	-	-	-	-
1	-	-	-	-	-	6	-	-	*r0*
2	-	-	-	-	-	*r2*	7	*r2*	*r2*
3	-	-	-	-	-	*r4*	*r4*	*r4*	*r4*
4	8	2	3	4	5	-	-	-	-
5	-	-	-	-	-	*r6*	*r6*	*r6*	*r6*
6	-	9	3	4	5	-	-	-	-
7	-	-	10	4	5	-	-	-	-
8	-	-	-	-	-	6	-	11	-
9	-	-	-	-	-	*r1*	7	*r1*	*r1*
10	-	-	-	-	-	*r3*	*r3*	*r3*	*r3*
11	-	-	-	-	-	*r5*	*r5*	*r5*	*r5*

Não havendo conflitos, a gramática é sLR(1). Vamos ver como seria a análise da cadeia x=(a+a)*a. Na configuração inicial, a pilha contem apenas o estado inicial 0; a entrada contem a cadeia x. Acrescentamos à pilha os símbolos correspondentes aos estados, que, naturalmente, devem ser entendidos apenas como comentários. As configurações sucessivas são:

Pilha	Entrada	*Ação*
0	(a+a)*a	empilhar: 4
0 (4	a+a)*a	empilhar: 5
0 (4 a 5	+a)*a	*reduzir: 6*
0 (4 F 3	+a)*a	*reduzir: 4*
0 (4 T 2	+a)*a	*reduzir: 2*

Pilha	Entrada	*Ação*
0 (4 E 8	+a)∗a	empilhar: 6
0 (4 E 8 + 6	a)∗a	empilhar: 5
0 (4 E 8 + 6 a 5	)∗a	*reduzir: 6*
0 (4 E 8 + 6 F 3	)∗a	*reduzir: 4*
0 (4 E 8 + 6 T 9	)∗a	*reduzir: 1*
0 (4 E 8	)∗a	empilhar: 11
0 (4 E 8) 11	∗a	*reduzir: 5*
0 F 3	∗a	*reduzir: 4*
0 T 2	∗a	empilhar: 7
0 T 2 ∗ 7	a	empilhar: 5
0 T 2 ∗ 7 a 5	ε	*reduzir: 6*
0 T 2 ∗ 7 F 10	ε	*reduzir: 3*
0 T 2	ε	*reduzir: 2*
0 E 1	ε	*reduzir: 0 (aceitar)*

A seqüência das reduções, como esperado, é 6 4 2 6 4 1 5 4 6 3 2 0, correspondente à derivação direita

$$S' \Rightarrow E \Rightarrow T \Rightarrow T*F \Rightarrow T*a \Rightarrow F*a \Rightarrow (E)*a \Rightarrow (E+T)*a \Rightarrow (E+F)*a$$
$$\Rightarrow (E+a)*a \Rightarrow (T+a)*a \Rightarrow (F+a)*a \Rightarrow (a+a)*a$$

□

Uma gramática ambígua não pode ser sLR(1), uma vez que a existência de duas derivações direitas para uma cadeia x implica em que algum ponto da construção dessas derivações por um analisador sLR(1) duas ações diferentes vão ser encontradas, cada uma correspondendo a uma derivação. Veja o Exemplo seguinte.

Exemplo 20: Considere a gramática do Exemplo 16, já aumentada:

```
0. S' →   S
1. S  →   if E th S el S
2.    |   if E th S
3.    |   s
4. E  →   e
```

Como observado anteriormente, esta gramática é ambígua, e, portanto, não é sLR(1). Este fato é verificado a seguir, pela construção da tabela de ações. Entretanto, considerando duas derivações de `if e th if e th s el s`, temos:

```
S ⇒ if E th S
  ⇒ if E th if E th S el S
  ⇒ if E th if E th S el s
  ⇒ if E th if E th s el s
  ⇒ if E th if e th s el s
  ⇒ if e th if e th s el s

S ⇒ if E th S el S
  ⇒ if E th if E th S el S
  ⇒ if E th if E th S el s
```

$\Rightarrow$ `if E th if E th s el s`
$\Rightarrow$ `if E th if e th s el s`
$\Rightarrow$ `if e th if e th s el s`

Portanto, no estado alcançável a partir do estado inicial por `if E th if E th S`, teremos duas ações possíveis com o símbolo `el`: empilhamento passando para o estado alcançável a partir do estado inicial por `if E th if E th S el`, e redução pela regra `S` $\rightarrow$ `if E th S`. Isso pode ser verificado pelo exame do estado 7 (alcançável via `if E th if E th S`), que apresenta um conflito com o símbolo `el` entre empilhamento para 8 e redução pela regra 2.

0.	S' $\rightarrow$ •S S $\rightarrow$ •if E th S el S S $\rightarrow$ •if E th S S $\rightarrow$ •s
1.	S' $\rightarrow$ S•
2.	S $\rightarrow$ if•E th S el S S $\rightarrow$ if•E th S E $\rightarrow$ •e
3.	S $\rightarrow$ s•
4.	S $\rightarrow$ if E•th S el S S $\rightarrow$ if E•th S
5.	E $\rightarrow$ e•
6.	S $\rightarrow$ if E th•S el S S $\rightarrow$ if E th•S S $\rightarrow$ •if E th S el S S $\rightarrow$ •if E th S S $\rightarrow$ •s
7.	S $\rightarrow$ if E th S•el S S $\rightarrow$ if E th S•
8.	S $\rightarrow$ if E th S el•S S $\rightarrow$ •if E th S el S S $\rightarrow$ •if E th S S $\rightarrow$ •s
9.	S $\rightarrow$ if E th S el S•

Temos Follow(S') = { $ }, Follow(S) = { $, el }, e Follow(E) = { th }. Portanto, a tabela de ações é

	S	E	if	th	el	s	e	$
0	1	-	2	-	-	3	-	-
1	-	-	-	-	-	-	-	*r0*
2	-	4	-	-	-	-	5	-
3	-	-	-	-	*r3*	-	-	*r3*
4	-	-	-	6	-	-	-	-
5	-	-	-	*r4*	-	-	-	-
6	7	-	2	-	-	3	-	-
7	-	-	-	-	8/*r2*	-	-	*r2*
8	9	-	2	-	-	3	-	-
9	-	-	-	-	*r1*	-	-	*r1*

Como no caso LL(1), a preferência na resolução do conflito é por considerar o `el` como parte do último `if` aberto. Neste caso, isto leva à eliminação da redução *r2*, dando preferência ao empilhamento do estado 8. Deixamos ao leitor a verificação da correção do analisador resultante. Um aspecto curioso é que a maioria dos analisadores ascendentes procura empilhar o símbolo de lookahead antes de tentar reduzir, ou seja, dá preferência automaticamente à escolha correta, neste caso.

□

Exercício 3: Mostre que a gramática abaixo é *(1)* equivalente à do Exemplo 20, *(2)* não ambígua, e *(3)* sLR(1).

```
S' → S
S → if E th Se el S  | if E th S | s
Se → if E th Se el Se | s
E → e
```

□

Exercício 4: A gramática a seguir é ambígua, e equivalente a gramática dos Exemplos 1 e 19. Construa um analisador semelhante ao sLR(1) para essa gramática, escolhendo, nos casos de conflitos, as ações corretas de acordo com as convenções usuais de precedência e de associatividade pela esquerda.

```
E → E + E | E * E | ( E ) | a
```

□

3.5 - Analisadores LR(1)

Vamos agora estudar o método LR(1) original, conhecido às vezes como LR(1) *canônico*, para distinguir de variantes como o método LR(1) *simples* (sLR(1), já visto), ou do método LR(1) *lookahead* (laLR(1), a ser apresentado na próxima seção). No método LR(1), a decisão sobre os símbolos de lookahead que permitem as reduções pelas várias regras é feita com um cuidado maior do que no caso sLR(1). No caso sLR(1), uma redução por uma regra $A \rightarrow \beta$ era feita para todos os símbolos de Follow(A). Isso significa que, num dado estado, podem estar sendo levados em consideração símbolos de lookahead que foram introduzidas por regras da gramática que não interferem nesse estado, ou seja, símbolos que não podem aparecer nessa posição em nenhuma cadeia correta que leve ao estado considerado. No analisador LR(1), os símbolos que podem ocorrer como lookaheads são calculados para cada caso, permitindo ao analisador LR(1) um controle mais fino sobre as reduções, o que faz com que algumas gramáticas que não são sLR(1) sejam LR(1), isto é, tenham analisadores LR(1) canônicos.

Para calcular os lookaheads admissíveis, usamos uma definição de item com mais informação: um item LR(1) é da forma $[A \rightarrow \alpha \bullet \beta, u]$, onde u é um símbolo terminal. (Lembramos que $ é tratado como terminal, em casos como este.) Na realidade, a teoria define um item LR(k), e u é uma cadeia com k símbolos; no nosso caso, k=1, e u é uma cadeia de um símbolo. Os itens usados na construção do analisador sLR(1) são itens LR(0), com a notação simplificada: em vez de $[A \rightarrow \alpha \bullet \beta, \varepsilon]$, escrevemos apenas $A \rightarrow \alpha \bullet \beta$. Na prática, valores de k maiores que 1 não são usados, porque a tabela de ações teria n^k colunas, onde n é o número de símbolos da gramática. Numa gramática de uma linguagem de programação podemos esperar pelo menos 300 símbolos, incluindo-se aí terminais e nãoterminais.

As regras para construção da coleção dos estados LR(1) são as seguintes:

- o fechamento de um estado q é feito acrescentando ao estado q, para cada item $[A \rightarrow \alpha \bullet B\beta, u]$ pertencente a q, todos os itens da forma $[B \rightarrow \bullet\gamma, v]$, onde $B \rightarrow \gamma$ é uma regra de B, e $v \in First(\beta u)$. Como antes, o processo deve ser repetido enquanto novos itens forem sendo acrescentados.

Note que os símbolos $v \in First(\beta u)$ são aqueles que podem ocorrer depois de B, supondo que u ocorreu depois de A, e que a regra $A \rightarrow \alpha B \beta$ foi usada.

- a transição entre estados é definida da seguinte maneira: se um estado q tem um ou mais itens da forma $[A \rightarrow \alpha \bullet X \beta, u]$, o estado $\delta(p,X)$, alcançado com o símbolo X a partir de p é obtido pelo fechamento do conjunto de itens da forma $[A \rightarrow \alpha X \bullet \beta, u]$, obtidos a partir de todos os itens com X depois do ponto em q.
- o estado inicial 0 é construído pelo fechamento do conjunto { [S'→•S, $] }.
- a coleção de estados do analisador sLR(1) é obtida a partir do estado inicial 0, através de todas as transições possíveis.

Os empilhamentos são feitos obedecendo à tabela de transições; as reduções são feitas de acordo com os símbolos constantes dos itens completos correspondentes às reduções, isto é, se um estado tem um item $[B \rightarrow \gamma \bullet, v]$, então a esse estado, quando o primeiro símbolo da entrada é v corresponde a ação de redução pela regra $B \rightarrow \gamma$.

Exemplo 21: Considere a gramática do Exemplo 1. O estado inicial LR(1) para essa gramática tem 17 itens:

```
0.{ [S'→•E, $],   [E→•E+T, $],   [E→•T, $],   [E→•E+T, +],
    [E→•T, +],    [T→•T*F, $],   [T→•F, $],   [T→•T*F, +],
    [T→•F, +],    [T→•T*F, *],   [T→•F, *],   [F→•(E), $],
    [F→•a, $],    [F→•(E), +],   [F→•a, +],   [F→•(E), *],
    [F→•a, *] }
```

Vamos abreviar os estados, reunindo os diversos lookaheads correspondentes ao mesmo item LR(0) Assim, o estado 0 será representado por

```
0.{ [S'→•E, $],   [E→•E+T, $+],   [E→•T, $+],
                  [T→•T*F, $+*],  [T→•F, $+*],
                  [F→•(E), $+*],  [F→•a, $+*] }
```

Além disso, vamos anotar ao lado dos estados as transições e as reduções correspondentes. Usando essas convenções, a coleção de estados LR(1) e a função de transição são dadas por

0.	[S'→•E, $] [E→•E+T, $+] [E→•T, $+] [T→•T*F, $+*] [T→•F, $+*] [F→•(E), $+*] [F→•a, $+*]	E 1 T 2 F 3 (4 a 5
1.	[S'→E•, $] [E→E•+T, $+]	*r0* + 6
2.	[E→T•, $+] [T→T•*F, $+*]	*r2* * 7
3.	[T→F•, $+*]	*r4*
4.	[F→(•E), $+*] [E→•E+T,)+] [E→•T,)+] [T→•T*F,)+*] [T→•F,)+*] [F→•(E),)+*] [F→•a,)+*]	E 8 T 9 F 10 (11 a 12
5.	[F→a•, $+*]	*r6*
6.	[E→E+•T, $+] [T→•T*F, $+*] [T→•F, $+*] [F→•(E), $+*] [F→•a, $+*]	T 13 F 3 (4 a 5

7.	[T→T∗•F, $+∗]	F 14
	[F→•(E), $+∗]	(4
	[F→•a, $+∗]	a 5
8.	[F→(E•), $+∗]	) 15
	[E→E•+T,)+]	+ 16
9.	[E→T•,)+]	*r2*
	[T→T•∗F,)+∗]	∗ 17
10.	[T→F•,)+∗]	*r4*
11.	[F→(•E),)+∗]	E 18
	[E→•E+T,)+]	
	[E→•T,)+]	T 9
	[T→•T∗F,)+∗]	
	[T→•F,)+∗]	F 10
	[F→•(E),)+∗]	(11
	[F→•a,)+∗]	a 12
12.	[F→a•,)+∗]	*r6*
13.	[E→E+T•, $+]	*r1*
	[T→T•∗F, $+∗]	∗ 7
14.	[T→T∗F•, $+∗]	*r3*
15.	[F→(E)•, $+∗]	*r5*
16.	[E→E+•T,)+]	T 19
	[T→•T∗F,)+∗]	
	[T→•F,)+∗]	F 10
	[F→•(E),)+∗]	(11
	[F→•a,)+∗]	a 12
17.	[T→T∗•F,)+∗]	F 20
	[F→•(E),)+∗]	(11
	[F→•a,)+∗]	a 12
18.	[F→(E•),)+∗]	) 21
	[E→E•+T,)+]	+ 15
19.	[E→E+T•,)+]	*r1*
	[T→T•∗F,)+∗]	∗ 16
20.	[T→T∗F•,)+∗]	*r3*
21.	[F→(E)•,)+∗]	*r5*

As ações de empilhamento e de redução estão anotadas na tabela acima. Por exemplo, o estado 19 tem 5 itens: os dois primeiros ([E→E+T•,)] e [E→E+T•, +]), são itens completos, e indicam a redução pela regra 1 (E→E+T) para) e +; os outros três ([T→T•∗F,)], [T→T•∗F, +] e [T→T•∗F, ∗]) indicam uma ação de empilhamento do símbolo ∗ com transição para o estado 16, que contém os três itens resultantes ([T→T∗•F,)], [T→T∗•F, +] e [T→T∗•F, ∗]). A tabela de ações pode ser construída diretamente da tabela acima.

Note que o analisador LR(1) para esta gramática tem quase o dobro do número de estados do analisador sLR(1) para a mesma gramática. Isto ocorre porque a maioria dos estados sLR(1) se dividiu em dois, um correspondente ao uso da construção dentro de parênteses (o símbolo) aparece como lookahead), e outro fora dos parênteses (o símbolo $ aparece como lookahead). Essa discriminação é que faz com que este método de análise seja mais geral, isto é, que possa ser aplicado a mais gramáticas. Note entretanto que, por construção, todo símbolo que aparece como lookahead pertence ao Follow do nãoterminal correspondente.

Mostramos a seguir os passos na aceitação de (a+a)∗a. Os símbolos indicados na pilha devem ser entendidos como comentários: servem apenas para indicar os símbolos correspondentes aos estados empilhados.

Pilha	Entrada	Ação
0	(a+a)∗a	s4
0 (4	a+a)∗a	s12
0 (4 a 12	+a)∗a	*r6*
0 (4 F 10	+a)∗a	*r4*
0 (4 T 9	+a)∗a	*r2*

Pilha	Entrada	Ação
0 (4 E 8	+a)∗a	s16
0 (4 E 8 + 16	a)∗a	s12
0 (4 E 8 + 16 a 12	)∗a	*r6*
0 (4 E 8 + 16 F 10	)∗a	*r4*
0 (4 E 8 + 16 T 19	)∗a	*r1*
0 (4 E 8	)∗a	s15
0 (4 E 8) 15	∗a	*r5*
0 F 3	∗a	*r4*
0 T 2	∗a	s7
0 T 2 ∗ 7	a	s5
0 T 2 ∗ 7 a 5		*r6*
0 T 2 ∗ 7 F 14		*r3*
0 T 2		*r2*
0 E 1		*r0*

□

Exemplo 22: Considere a gramática

```
S → Cbc
  | Dbd
C → a
D → a
```

Essa gramática não é LR(1), apesar de não ser ambígua, e de gerar uma linguagem com apenas duas cadeias: `abc` e `abd`. A razão para isso é que o segundo símbolo não traz informação suficiente para decidir se o primeiro símbolo (`a`) deve ser reduzido para `C` ou `D`. Para isso é necessário olhar o terceiro símbolo: se for um `c`, a redução deve ser feita para `C`; se for um `d`, a redução deve ser feita para `D`.

Deixamos como exercício a verificação da existência do conflito. Observamos que a linguagem é LR(1), isto é, a linguagem tem uma gramática LR(1),

```
S → abc
  | abd
```

□

Exemplo 23: Considere a linguagem

$\{ a^i b^i c \mid i \ 0\} \cup \{ a^i b^{2i} d \mid i \geq 0 \}$.

Esta linguagem é livre de contexto, como se pode ver pela gramática

```
S → S1 c
  | S2 d
S1 → a S1 b
  | ε
S2 → a S2 b b
  | ε
```

Esta gramática não é ambígua, mas também não é LR(1). Considere, por exemplo, as cadeias a^3b^3c e a^3b^6d. Quando o primeiro b é encontrado, não há informação suficiente para um analisador LR(1) decidir se a redução deve ser feita pela regra $S1 \rightarrow \varepsilon$ ou pela regra $S2 \rightarrow \varepsilon$. Não há ambigüidade: a informação necessária está contida no último símbolo (c ou d).

Na realidade, isto (ou alguma coisa parecida) acontece com qualquer outra gramática dessa linguagem: ao encontrar o primeiro b, é impossível decidir qual a regra a ser usada, se uma regra que associa um b a cada a, ou uma regra que associa dois b's a cada a. Linguagens como esta são chamadas *não determinísticas*, e não têm gramáticas LR(1).

Observamos finalmente que nenhum projetista de linguagens de programação introduz intencionalmente construções não determinísticas. Ao contrário, a preocupação é sempre no sentido de organizar o programa de forma que as informações importantes sempre estejam disponíveis antes do ponto em que se tornam necessárias. Por exemplo, as declarações precedem os comandos cuja análise utiliza as informações por elas introduzidas.

□

3.6 - Analisadores laLR(1)

O método de análise LR(1) é um método bastante geral, e poderia ser sempre usado, não fosse o número grande de estados gerados, que faz com que as tabelas LR(1) ocupem um espaço considerável. Por exemplo, uma gramática de Pascal LR(1) tem uns 3000 estados, enquanto em uma gramática sLR(1) de Pascal temos um décimo disso: uns 300 estados. Entretanto, é mais difícil encontrar gramáticas sLR(1) do que gramáticas LR(1), para as linguagens de programação usuais. Isto levou os pesquisadores a procurar um método de análise que calculasse os símbolos de lookahead, como é feito pelo LR(1), mas que levasse a um número menor de estados, como o sLR(1). Esse método é o método laLR(1), que já se encontra em uso há uns 20 anos, popularizado pelo gerador de analisadores sintáticos *yacc*.

A idéia do método laLR(1) é simples. Para os propósitos desta discussão, defina o *núcleo*[2] de um conjunto de itens LR(1) como sendo o conjunto de itens LR(0) obtido retirando todos os lookaheads de todos os itens.

- Construa um analisador LR(1), e identifique todos os estados que têm o mesmo núcleo.
- Para cada grupo de estados LR(1) $\{ p_1, p_2, ..., p_n \}$ que têm o mesmo núcleo, construa um estado laLR(1) p pela união de todos esses estados: $p=p_1 \cup p_2 \cup ... \cup p_n$.

O estado união p tem o mesmo núcleo que $p_1, p_2, ..., p_n$.

Feito isso, a função de transição laLR(1) pode ser construída: se $p=p_1 \cup p_2 \cup ... \cup p_n$, determine $\delta(p, X)$ da seguinte maneira:

[2] *cerne* seria uma tradução melhor do inglês *kernel*.

- escolha um dos estados p_i que fazem parte de p, e obtenha $q_i = \delta(p_i, X)$
- determine de qual estado laLR(1) q o estado q_i faz parte.
- faça $\delta(p, X)=q$.

Para determinar as reduções, a regra é a mesma do analisador LR(1): se um estado tem um item $[B \rightarrow \gamma\bullet, v]$, então a esse estado, quando o primeiro símbolo da entrada é v corresponde a ação de redução pela regra $B \rightarrow \gamma$.

O Exemplo a seguir mostra os detalhes da construção.

Exemplo 24: Considere a gramática do Exemplo 1, cujo analisador LR(1) aparece no Exemplo 21. Podemos identificar os estados LR(1) que devem ser reunidos na construção do analisador laLR(1). São os pares 2-9, 3-10, 4-11, 5-12, 6-16, 7-17, 8-18, 13-19, 14-20, 15-21. Por exemplo, o estado 2-9 terá os 5 itens { [E→T•, $+)], [T→T•*F, $+*)] }. Do estado 2-9 temos uma transição com * para 7-11, e redução pela regra 2 com $, + e). A linha da tabela de ação correspondente ao estado 2-9:

	E	T	F	(	a	+	*	)	$
2-9	-	-	-	-	-	*r2*	7-17	*r2*	*r2*

Outra possibilidade é reunir as linhas dos estados componentes 2 e 9, que são

	E	T	F	(	a	+	*	)	$
2	-	-	-	-	-	*r2*	7	-	*r2*
9	-	-	-	-	-	*r2*	17	*r2*	-

Deixamos a construção do restante da tabela como exercício. Observamos apenas que, para a gramática do exemplo, a tabela resultante é equivalente à tabela do analisador sLR(1) porque em todos os casos, após a união teremos como lookaheads exatamente os símbolos pertencentes aos conjuntos Follow dos nãoterminais correspondentes.

□

Podemos também construir a coleção de estados laLR(1) para uma gramática, sem passar pela coleção de estados LR(1). Primeiro, construímos os estados LR(0) para a gramática, e a correspondente função de transição. (Estes são exatamente aos estados e as transições do analisador sLR(1).) Depois disso, acrescentamos o item [S'→S, $] ao estado inicial 0, e acrescentamos o estado 0 à lista dos estados a serem tratados. (Note que o núcleo do item já faz parte do estado LR(0), e só vamos agora acrescentar os lookaheads.) *Tratar* um estado q significa

- remover o estado q da lista.
- fazer o fechamento do estado q, propagando os lookaheads;
- para cada item $[A \rightarrow \alpha\bullet X\beta, u]$, verificar se no estado $q' = \delta(q, X)$ já se encontra o item $[A \rightarrow \alpha X\bullet\beta, u]$. Se não for encontrado, o item é acrescentado, e se o estado q' não faz parte da lista, é acrescentado à lista, para tratamento posterior.

Um estado pode ser tratado várias vezes; em particular, durante o tratamento de um estado q ele próprio pode ser re-incluído na lista. O processo continua até que a lista fique vazia, ou seja, até que não haja mais nenhum estado para ser tratado.

Exemplo 25: Considere a gramática aumentada

```
0.   S' → E
1.   E → E + P
2.   E → P
3.   P → if B th P el P
4.   P → a
5.   B → E = E
6.   B → if B th B el B
```

Vamos construir a coleção de estados laLR(1), sem passar pela coleção de estados LR(1). Primeiro, construímos a tabela de estados e de transições LR(0).

O resultado deste processo está indicado na tabela a seguir.

0.	S'→•E E→•E + P E→•P P→•if B th P el P P→•a	$ $ + $ + $ + $ +	E 1 P 2 if 3 a 4
1.	S'→E• E→E•+ P	$ $ +	*r0* + 5
2.	E→P•	$ + = th el	*r2*
3.	P→if•B th P el P B→•E = E B→•if B th B el B E→•E + P E→•P P→•if B th P el P P→•a	$ + = th el th th = + = + = + = +	B 6 E 7 if 8 P 2 a 4
4.	P→a•	$ + = th el	*r4*
5.	E→E +•P P→•if B th P el P P→•a	$ + = th el $ + = th el $ + = th el	P 9 if 3 a 4
6.	P→if B•th P el P	$ + = th el	th 10
7.	B→E•= E E→E•+ P	th el = +	= 11 + 5
8.	B→if•B th B el B P→if•B th P el P B→•E = E B→•if B th B el B E→•E + P E→•P P→•if B th P el P P→•a	th el = + el th th = + = + = + = +	B 12 E 7 if 8 P 2 a 4
9.	E→E + P•	$ + = th el	*r1*

10. P→if B th•P el P P→•if B th P el P P→•a	$ + = th el el el	P 13 if 3 a 4
11. B→E =•E E→•E + P E→•P P→•if B th P el P P→•a	th el th + el th + el th + el th + el	E 14 P 2 if 3 a 4
12. B→if B•th B el B P→if B•th P el P	th el = + el	th 15
13. P→if B th P•el P	$ + = th el	el 16
14. B→E =•E E→•E + P	th el th + el	*r5* + 5
15. B→if B th•B el B P→if B th P•el P B→•E = E B→•if B th B el B P→•if B th P el P P→•a E→•E + P E→•P	th el = + el el el el el = + = +	B 17 P 18 E 7 if 8 a 4
16. P→if B th P el•P P→•if B th P el P P→•a	$ + = th el $ + = th el $ + = th el	P 19 if 3 a 4
17. B→if B th B•el B	th el	el 20
18. P→if B th P•el P E→P•	= + el = +	el 16 *r2*
19. P→if B th P•el P	$ + = th el	*r3*
20. B→if B th B el•B B→•E = E B→•if B th B el B E→•E + P E→•P P→•if B th P el P P→•a	th el th el th el = + = + = + = +	B 21 E 7 if 8 P 2 a 4
21. B→if B th B el B•	th el	*r6*

Inicialmente, o estado 0 só contém o item [S'→•E, $]. Por fechamento, são acrescentados os itens [E→•E + P, $+], [E→•P, $+], [P→•if B th P el P, $+] e [P→•a, $+]. Por causa desses itens, o estado 1 recebe os itens [S'→E•, $] e [E→E•+P, $+], o estado 2 recebe os itens [E→P•, $+], o estado 3 recebe os itens [P→if•B th P el P, $+], o estado 4 recebe os itens [P→a•, $+] e todos esses estados entram na lista de estados a serem tratados. Note que esses estados recebem outros itens de outras transições. Por exemplo, o item com lookahead = no estado 2 é proveniente de uma transição a partir do estado 3.

Note que esta gramática não é sLR(1). Se, no estado 18, substituirmos { =, + } por Follow(E) = { $, +, =, th, el }, teremos um conflito do tipo empilha/reduz com el.

18. P→if B th P•el P	= + el	el 16
E→P•	= +	*r2*

□

Exercício 5: Prove as afirmativas a seguir:

- a gramática abaixo é LR(1), mas não é sLR(1) ou laLR(1).

S → (A) | (B] | [B) | [A]
A → a
B → a

- a gramática abaixo é LR(1) e laLR(1), mas não é sLR(1).

S → (A) | (B] | [B) | { A]
A → a
B → a

□

Observamos finalmente que, pelas próprias definições, toda gramática sLR(1) é laLR(1), e toda gramática laLR(1) é LR(1). Resultados não demonstrados aqui mostram que toda gramática LL(1) é LR(1). Em termos de linguagens, as classes sLR(1), laLR(1) e LR(1) são equivalentes. Entretanto, existem linguagens LR(1) que não têm gramáticas LL(1).

(rev. mar 99)