3. ANÁLISE SINTÁTICA

INTRODUÇÃO

Toda linguagem de programação tem regras que descrevem sua estrutura sintática (ou sintaxe). A sintaxe de uma LP pode ser descrita por uma gramática livre de contexto[1] ou pela notação BNF.

O uso de gramáticas traz vantagens para projetistas de linguagens e escritores de compiladores pelas seguintes razões:

- uma gramática dá uma especificação sintática precisa de uma LP (embora talvez menos clara que a notação BNF);
- para certas classes de gramáticas, pode-se automatizar o processo de construção do analisador sintático e o gerador automático pode revelar certas ambigüidades sintáticas da LP difíceis de serem detectadas pelo escritor do compilador;
- uma gramática bem projetada dá estrutura à uma LP, o que facilita a compilação e a detecção de erros de programas fonte; e
- novas construções sintáticas que surgem com a evolução de uma LP podem ser incorporadas mais facilmente à linguagem se seu compilador tem uma implementação baseada em uma descrição gramatical.

[1]Gramática livre de contexto é aquela que tem regras do tipo $A ::= \alpha$, com $A \in VN$ e $\alpha \in V^*$

Dada uma gramática G(S), verificar se uma dada sentença *w* pertence ou não à L(G) é verificar se

$$S \Rightarrow^+ w$$

para alguma seqüência de derivação

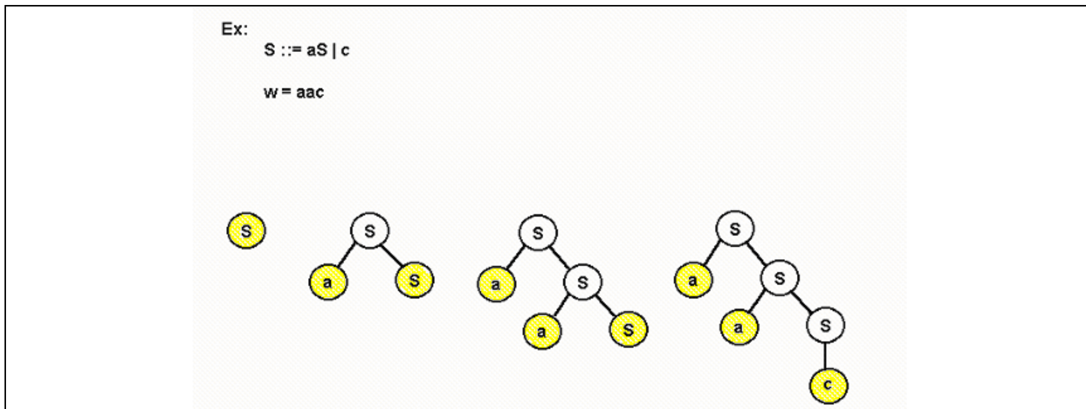
$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \Rightarrow w$$

(em outra palavras, é obter uma árvore de derivação sintática (ADS) para *w*).

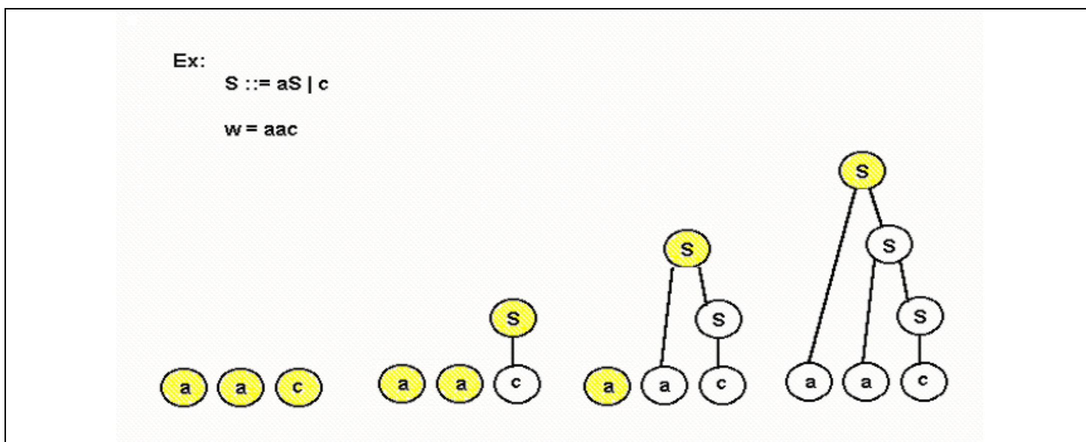
Nesse sentido, o analisador sintático de um compilador nada mais é do que um construtor de ADS. Quando ele consegue construir uma ADS para uma sentença *w* (um programa), dizemos que *w* está sintaticamente correta. Em caso contrário, dizemos que *w* está sintaticamente incorreta.

Os analisadores sintáticos podem ser classificados basicamente em dois grupos: descendentes e ascendentes.

Um *analisador sintático descendente* tenta construir a ADS para uma sentença w a partir do símbolo inicial S (raiz), aplicando regras de produção até produzir todos os símbolos (folhas) de w.



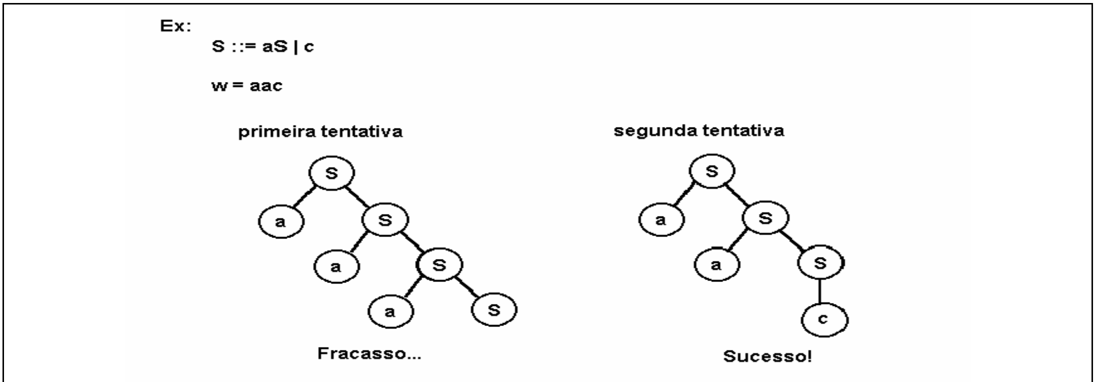
Um *analisador sintático ascendente* tenta construir a ADS para uma sentença *w* a partir dos símbolos de *w* (folhas), fazendo reduções (substituir o lado direito de uma regra pelo seu lado esquerdo) até obter o símbolo inicial *S* (raiz).



ANALISADOR SINTÁTICO DESCENDENTE

Descendente com Backup

O *analisador sintático descendente com backup* é o mais antigo método de construção de ADS. Usando tentativa-e-erro, tenta derivar uma sentença esgotando todas as possíveis opções de derivação (p.ex. várias regras com o mesmo lado esquerdo, A ::= aB | aC | cD). Caso uma escolha tenha sido infeliz, é selecionada outra derivação e o processo continua. O retorno para os pontos com outras possibilidades de derivação é feito até que se tenha analisado a sentença inteira, ou até que seja encontrada uma folha que não seja reconhecida depois de esgotar-se todas as regras de produção da gramática.



O registro de que regra de produção foi adotada em determinado ponto da análise, que porção da sentença analisada já foi lida, etc. é mantido em uma estrutura tipo pilha para facilitar o mecanismo de tentativa-e-erro do método descendente com backup.

Por gastar muita memória para registrar os passos que adota durante a análise de uma sentença, além de ser demorado, este método praticamente não é mais usado na construção de compiladores.

Descendente Recursivo

Para quem dispõe de uma linguagem com recursividade para a implementação de um compilador, este método é o mais indicado pois aproveita a estrutura da gramática de forma completa e não impõe restrições à ela (a única restrição é a de não haver recursividade a esquerda).

O analisador sintático recursivo descendente é escrito na forma de um conjunto de procedimentos, sendo associado a cada procedimento um elemento não-terminal da gramática.

Por exemplo, dada a gramática:

<expr> ::= <termo> + <expr> | <termo>
<termo> ::= <fator> * <termo> | <fator>
<fator> ::= <primário> ** <fator> | <primário>
<primário> ::= IDENT | NÚMERO | (<expr>)

podemos reescrevê-la na forma abaixo, usando uma meta-linguagem onde *{w}+* significa uma ou mais ocorrências de *w* e *{w}** significa zero ou mais ocorrências de w:

(1) <expr> ::= <termo> { + <expr> }*
(2) <termo> ::= <fator> { * <termo> }*
(3) <fator> ::= <primário> { ** <fator> }*
(4) <primário> ::= IDENT | NÚMERO | (<expr>)

Podemos interpretar a regra (1) como: "expressão é um termo seguido de zero ou mais expressões conectadas pelo operador +", ou seja:

expressão = termo
expressão = termo + expressão
expressão = termo + expressão + expressão

e assim por diante.

Podemos, agora, escrever um algoritmo que faça a análise de sentenças geradas a partir dessa gramática. Vejamos como logo a seguir.

```
Procedimento analisador sintático {
    obtenha_símbolo(); /* chama o léxico */
    EXPR();
}

Procedimento EXPR() {
    TERMO();
    se símbolo_lido = '+' então {
        obtenha_símbolo();
        EXPR();
    }
}

Procedimento TERMO() {
    FATOR();
    se símbolo_lido = '*' então {
        obtenha_símbolo();
        TERMO();
    }
}
```

```
Procedimento FATOR() {
     PRIMÁRIO();
     se símbolo_lido = '**' então {
          obtenha_símbolo();
          FATOR();
     }
}

Procedimento PRIMÁRIO() {
     se símbolo_lido = IDENT então {
          /* trate adequadamente um identificador */
          obtenha_símbolo();
     }
     senão se símbolo_lido = NÚMERO então {
          /* trate adequadamente um número */
          obtenha_símbolo();
     }
     senão se símbolo_lido = '(' então {
          obtenha_símbolo();
          EXPR();
          se símbolo ≠ ')' então
               ERRO( "falta )" );
          senão
               obtenha_símbolo();
     }
}
```


PROBLEMA DE RECURSIVIDADE

Para que se possa construir um analisador sintático descendente, uma gramática não pode ter regras recursivas à esquerda (diretas ou indiretas). Uma gramática é recursiva à esquerda se tem produções da forma:

$U ::= U\alpha$ ou $U \Rightarrow^{+} U\alpha$

Nesse caso, qualquer algoritmo que implemente um analisador descendente vai entrar em ciclo ("loop") infinito.

Podemos resolver o problema de recursividade direta à esquerda com uma transformação muito simples na gramática. Regras do tipo:

$A ::= A\alpha \mid \beta$

cujo objetivo é produzir cadeias da forma:

$\beta, \beta\alpha, \beta\alpha\alpha, ...$

devem ser transformadas em:

$A ::= \beta A'$
$A' ::= \alpha A' \mid \&$

Por exemplo, dada a gramática:

<expr> ::= <expr> + <termo> | <termo>
<termo> ::= <termo> * <fator> | <fator>
<fator> ::= IDENT | (<expr>)

podemos produzir a versão:

<expr> ::= <termo> <expr'>
<expr'> ::= + <termo> <expr'> | &
<termo> ::= <fator> <termo'>
<termo'> ::= * <fator> <termo'> | &
<fator> ::= IDENT | (<expr>)

que não mais contém recursividade à esquerda (na verdade a recursividade foi transferida para a direita).

Descendente Preditor ou Analisador de Gramáticas LL(K)

Nós vimos que o analisador sintático descendente recursivo tem algumas restrições para sua implementação. Uma delas, já abordada anteriormente, diz respeito ao uso de produções recursivas à esquerda (diretas ou indiretas) que levam o analisador à um ciclo ("loop") infinito.

Outra restrição diz respeito à ocorrências de produções do tipo:

(1) $A ::= \alpha\beta$
(2) $A ::= \alpha\gamma$

que conduzem à uma situação onde, a partir do ponto A na árvore de derivação sintática de uma dada sentença, podemos derivar pela regra (1) ou pela regra (2) para se chegar à mesma cadeia α; ou seja, podemos aplicar mais de uma regra para chegarmos ao mesmo resultado.

Como resolver esse problema? Usando um desdobramento da regra de produção como se segue:

(3) $A ::= \alpha C$
(4) $C ::= \beta \mid \gamma$

Problema resolvido? Sim, podemos implementar o analisador sintático recursivo para quaisquer gramáticas que obedeçam essas duas restrições.

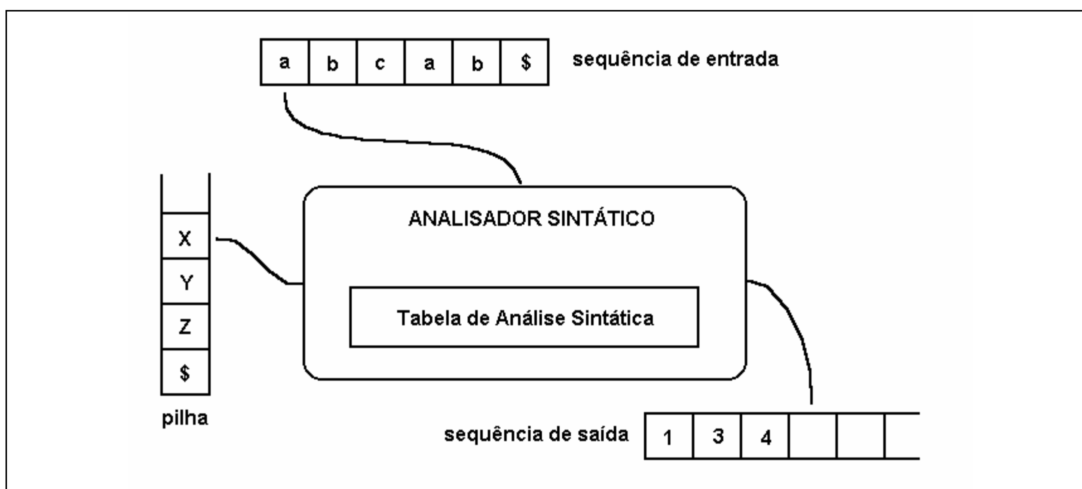
E se não temos disponível uma linguagem de programação recursiva p/ implementar o analisador? Nesse caso podemos utilizar um analisador sintático descendente preditor (ou analisador de gramáticas LL(K)).

A idéia do analisador LL(K) ("Left-to-right Left-most-derivation K") é de que basta olharmos no máximo K símbolos à frente na sentença, a partir do ponto em que estamos na ADS, para que possamos decidir que regra de produção aplicar.

```
Exemplo:

    S ::= aS | bS | c           S ::= abS | acS | ad

    G(S) é LL(1)                G(S) é LL(2)

    w = abc                     w = abad
    w = bac                     w = acad
```

Em termos de linguagens de programação, quase sempre é possível obter-se uma gramática LL(1) que permita o reconhecimento sintático de programas através de um analisador LL(1) que é bastante simples de implementar.

A idéia é a seguinte: o analisador sintático receberá uma seqüência de entrada (a sentença a ser analisada), manipulará uma estrutura de dados tipo pilha (onde monta a ADS), consultará uma tabela de análise sintática (tabela de "parsing") e emitirá uma seqüência de saída (regras que estão sendo aplicadas). Veja o esquema a seguir.



A seqüência de entrada é formada pela sentença a ser analisada, seguida por um símbolo delimitador ($).

A pilha contém uma seqüência de símbolos da gramática, precedida pelo indicador de base de pilha ($).

A tabela de análise sintática é uma matriz M[A,a] onde 'A' é um não-terminal e 'a' é um terminal ou dólar ($).

A seqüência de saída constará das produções aplicadas a partir do símbolo inicial (S), na geração da sentença.

Inicialmente a pilha contém o símbolo inicial da gramática precedido por dólar ($).

O analisador sintático, a partir de X, símbolo do topo da pilha, e próximo_símbolo, o atual símbolo da entrada, determina sua ação que pode ser uma das quatro possibilidades a seguir:

1) Se X é um terminal = próximo_símbolo = $, o analisador encerra sua atividade e comunica fim da análise sintática com sucesso;

2) Se X é um terminal = próximo_símbolo ≠ $, o analisador elimina X do topo da pilha e avança para o próximo símbolo de entrada;

3) Se X é um terminal ≠ próximo_símbolo, o analisador acusa um erro de sintaxe (chama rotina de tratamento de erros);

4) Se X é um não-terminal, o analisador consulta M[X, próximo_símbolo]. Se a resposta for uma regra de produção X ::= MVU, o analisador desempilha X do topo da pilha e empilha UVM (com M no topo da pilha). Para a saída é enviada a regra de produção usada. Se M[X, próximo_símbolo] = ERRO, o analisador acusa um erro de sintaxe (chama rotina de tratamento de erro).

Exemplo:

Seja a gramática G abaixo com a respectiva tabela de análise sintática.

(1) S ::= aAS (3) A ::= a
(2) S ::= b (4) A ::= bSA

	a	b	$
S	1	2	ERRO
A	3	4	ERRO

Dada a sentença w = abbab$, o analisador sintático assumiria as seguintes configurações durante a análise:

ENTRADA	PILHA	SAÍDA
abbab$	$S	
abbab$	$SAa	1
bbab$	$SA	1
bbab$	$SASb	1 4
bab$	$SAS	1 4
bab$	$SAb	1 4 2
ab$	$SA	1 4 2
ab$	$Sa	1 4 2 3
b$	$S	1 4 2 3
b$	$b	1 4 2 3 2
$	$	1 4 2 3 2

Vejamos o algoritmo do analisador preditor.

```
início
    /* seja X o símbolo do topo da pilha e
       próximo_símbolo o símbolo atual da entrada */

    enquanto X ≠ $ faça {
        se X é terminal então
            se X = próximo_símbolo então {
                elimine X do topo da pilha;
                leia_próximo_símbolo();
            }
            senão ERRO();
        senão se M[X, próximo_símbolo] = "X ::= Y1Y2...Yk então {
            elimine X do topo da pilha;
            empilhe Yk, ..., Y2, Y1
        }
        senão ERRO();
    }

    se próximo_símbolo ≠ $ então ERRO();
fim
```

A idéia e o algoritmo do preditor são bastante simples. Olhando com mais calma, porém, vemos que está faltando uma coisa fundamental. Como obter a tabela (ou matriz) de análise?

Para chegarmos até ela, precisamos introduzir dois novos conceitos (ou relações) em gramáticas. São os conceitos de Primeiro ("First") e Seguidor ("Follow"). Vejamos as definições a seguir.

Primeiro(α) = { x | α =>* xβ, com $\alpha \in$ V+, $\beta \in$ V*, x $\in$ VT*; se α =>* & então & $\in$ P(α) }

Seguidor(A) = { a | S =>* αAβ e a $\in$ P(β), com A $\in$ VN, α,β $\in$ V*, S símbolo inicial; se & $\in$ P(β) então S(β) $\in$ S(A), se S =>* αA então & $\in$ S(A) }

```
Exemplo:

    Dada a gramática:

        E   ::= TE'
        E'  ::= +TE' | &
        T   ::= FT'
        T'  ::= *FT' | &
        F   ::= ( E ) | id

Temos:

    Primeiro(E)    = { (, id }
    Primeiro(E')   = { +, & }
    Primeiro(T)    = { (, id }
    Primeiro(T')   = { *, & }
    Primeiro(F)    = { (, id }

    Seguidor(E     = Seguidor(E') = { ), & }
    Seguidor(T)    = Seguidor(T') = { +, ), & }
    Seguidor(F)    = { +, *, ), & }
```

Podemos então, ver o algoritmo para a geração da tabela (matriz) de análise sintática.


```
Algoritmo p/ obtenção da tabela de análise sintática

início
    para cada produção A ::= α da gramática, faça {
        para cada símbolo terminal a ∈ Primeiro(α), faça {
            adicione a produção A ::= α em M[A, a];
        }
        se & ∈ P(α), adicione a produção A ::= α
            em M[A, b], para cada terminal b ∈ Seguidor(A);

        se & ∈ P(α) e & ∈ Seguidor(A), adicione a
            produção A ::= α em M[A, $]
    }

    indique situação de ERRO para todas as posições
        indefinidas de M[A, a];
fim
```

Exemplo:

Aplicando-se o algoritmo dado para a gramática abaixo, obtém-se a tabela mostrada a seguir.

(1)	E	::= TE'		(5)	T'	::= *FT'
(2)	E'	::= +TE'		(6)	T'	::= &
(3)	E'	::= &		(7)	F	::= (E)
(4)	T	::= FT'		(8)	F	::= id

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

O algoritmo dado é válido para qualquer gramática, porém, para algumas gramáticas, a matriz M possui algumas entradas multiplamente definidas; por exemplo, se a gramática é recursiva à esquerda ou ambígua, temos pelo menos uma entrada multiplamente definida.

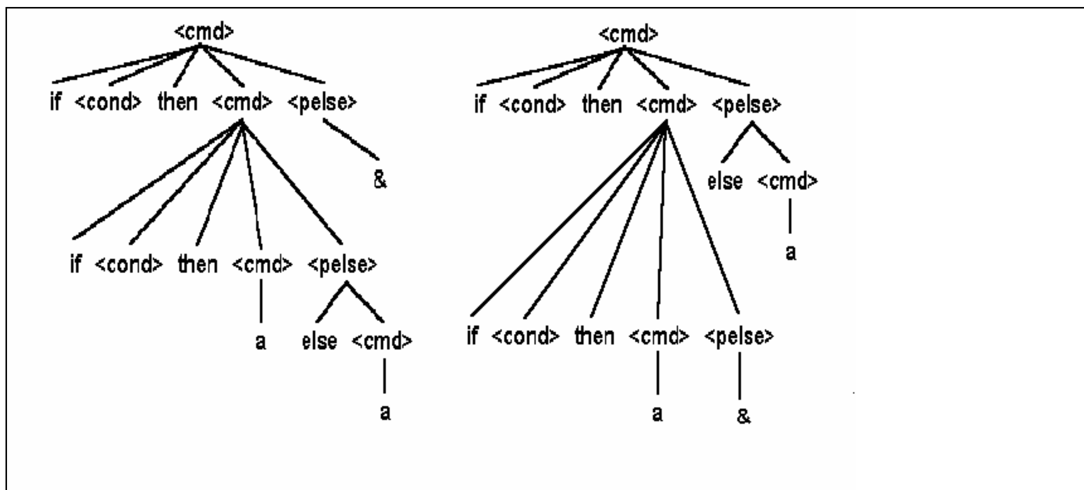
Exemplo: A gramática abaixo é ambígua para a sentença w, e pode ser interpretada de duas formas diferentes.

<cmd>	::= if <cond> then <cmd> <pelse>
<cmd>	::= a
<pelse>	::= else <cmd>
<pelse>	::= &
<cond>	::= b

W = if <cond> then if <cond> then a else a

```
        if <cond> then
            if <cond> then
                a
            else
                a

ou:

        if <cond> then
            if <cond> then
                a
        else
                a
```

Em termos de árvore de derivação sintática teríamos:



Para essa gramática, teríamos a seguinte matriz de análise sintática:

	a	b	if	then	else	$
<cmd>	2		1			
<pelse>					3 / 4	4
<cond>		5				

Gramáticas cujas tabelas de análise sintática não possuem entradas múltiplas definidas são ditas LL(1).

Implementação do Analisador Sintático Descendente LL(1)

Um analisador sintático preditor pode ser implementado facilmente utilizando-se certas convenções. A principal delas é a codificação de todos os símbolos usados na representação da gramática (terminais e não-terminais) através de números inteiros (os terminais podem/devem usar os códigos que foram atribuídos pelo analisador léxico e os não-terminais, para não serem confundidos com terminais, podem ser codificados através de números negativos).

Exemplo:

Dada a gramática:

(1)	E	::=	TE'	(5)	T'	::=	*FT'
(2)	E'	::=	+TE'	(6)	T'	::=	&
(3)	E'	::=	&	(7)	F	::=	(E)
(4)	T	::=	FT'	(8)	F	::=	id

podemos utilizar a seguinte codificação:

Símbolo	Código
id	1
+	11
*	13
(	21
)	22
$	99 /* fim de sentença/pilha */
E	-1
E'	-2
T	-3
T'	-4
F	-5

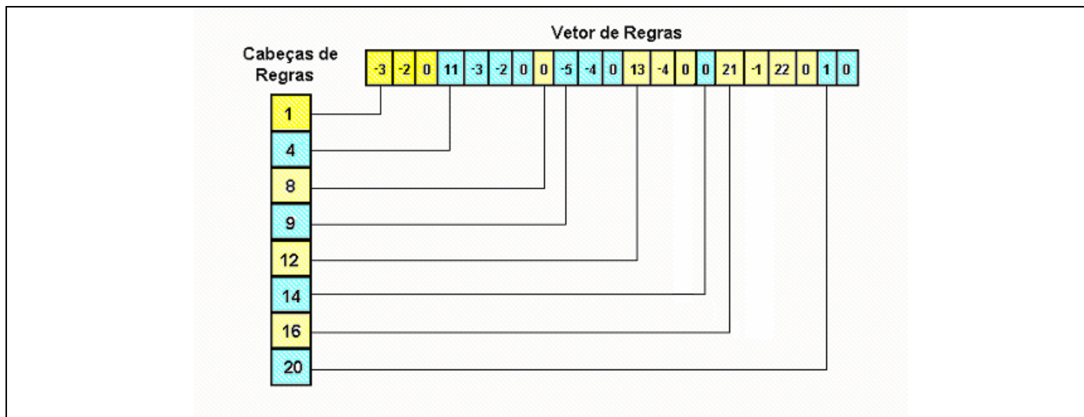
E as seguintes estruturas de dados:

- Pilha de análise: vetor de inteiros;
- Matriz de análise: matriz de inteiros;
- Vetor de regras: vetor de inteiros para conter todas as regras de produção codificadas.

A representação para a matriz de análise e o vetor de regras são indicadas a seguir.

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

OBS. As posições em branco são situações de erro.



O vetor de regras contém todas as regras de produção da gramática, já devidamente codificadas para facilitar o trabalho do analisador.

ANALISADOR SINTÁTICO ASCENDENTE

Como foi dito anteriormente, um *analisador sintático ascendente* tenta construir a árvore de derivação sintática (ADS) para uma sentença *w* a partir dos símbolos de *w* (folhas), fazendo reduções (substituir o lado direito de uma regra pelo seu lado esquerdo) até obter o símbolo inicial *S* (raiz).

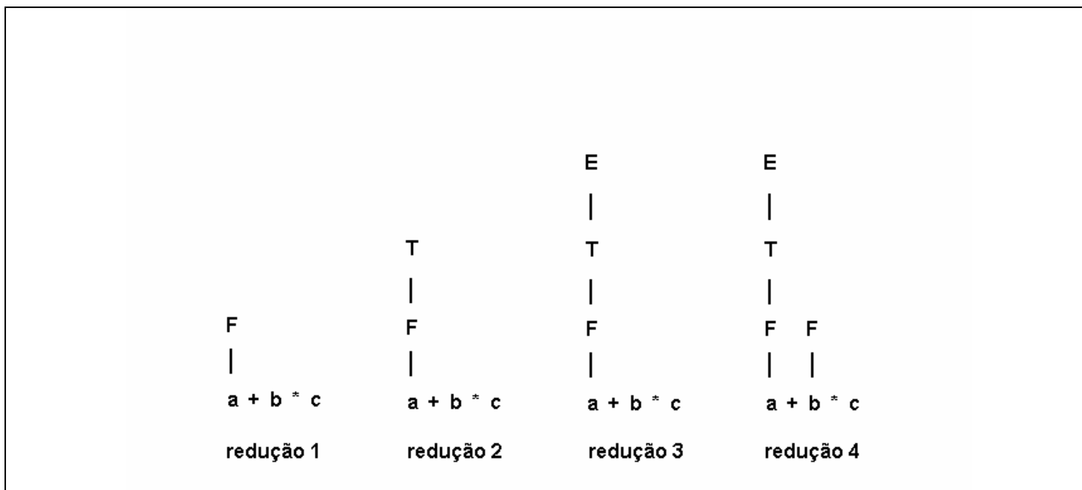
A idéia é basicamente a de procurar na forma sentencial corrente (que inicialmente é a sentença dada à análise) uma cadeia particular *u* tal que exista no conjunto de produções da gramática uma produção *U ::= u*. Encontrada a cadeia *u*, é feita a redução para o não terminal *U*, obtendo-se assim uma nova forma sentencial. Este processo é repetido até que seja obtido o símbolo inicial da gramática ou uma ocorrência de erro.

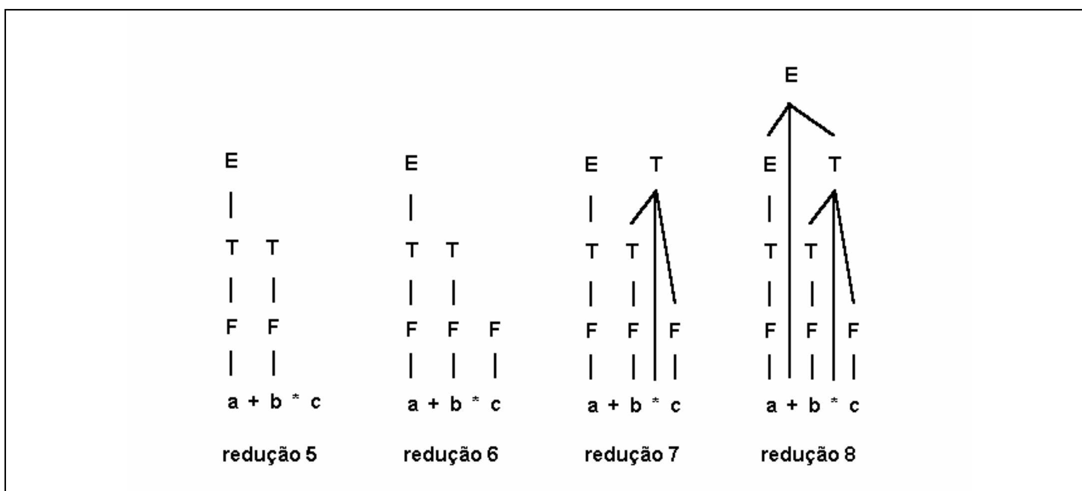
Exemplo:

Dada a gramática abaixo que gera expressões aritméticas:

E ::= E + T | T
T ::= T * F | F
F ::= a | b | c | (E)

e dada a sentença *w = a + b * c*, poderíamos ter as seguintes reduções no reconhecimento de *w*:


E
T
F
a + b * c
redução 1
redução 2
redução 3
redução 4


E
T
F
a + b * c
redução 5
redução 6
redução 7
redução 8

De posse da idéia de funcionamento do analisador ascendente, passemos a considerar os problemas que devem ser resolvidos:

1. Como identificar a parte da cadeia w que deve ser reduzida?

2. Que produção deve ser usada na redução?

Esses dois problemas podem ser facilmente resolvidos para um certo tipo de gramáticas denominadas Gramáticas de Precedência Simples.

GRAMÁTICAS DE PRECEDÊNCIA SIMPLES

Definição:

Seja *w = xuy* uma forma sentencial qualquer de uma gramática G. Nós dizemos que *u* é frase de *w* para um não terminal *U*, se:

S =>* xUy e U =>+ u, com $x, y \in V^*$ e $u \in V+$

Definição:

Particularmente, se:

U => u

chamamos *u* de frase simples de *w*.


```
Exemplo:

    Dada a gramática:

        <número>    ::= <nro>
        <nro>       ::= <nro> <dígito> | <dígito>
        <dígito>    ::= 0 | 1 | ... | 9

    Quais são as frases de w = <nro> 1?

        w = &   <nro>   1   &
            -   ---------   -
            x       u       y

        <número> =>*   &   <nro>   &   =>+
                       -   -----   -
                       x     U     y

                       &   <nro> 1     &
                       -   -------     -
                       x      u        y

    Logo, <nro> 1 é frase de w = <nro> 1.
```

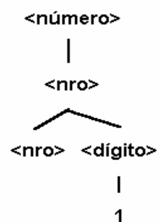
```
Exemplo (cont.):

    Considerando    <nro>   1   &
                    -----   -   -
                      x     u   y

        <número> =>*   <nro>   <dígito>    &   =>
                       -----   --------    -
                         x        U        y

                       <nro>   1   &
                       -----   -   -
                         x     u   y

temos que 1 é frase simples de w = <nro> 1.
```

Em termos de árvore de derivação sintática, *u* é uma frase se for uma cadeia de símbolos derivada de um único nó.

Exemplo:

Para a sentença do exemplo anterior, teríamos:

COMPILADORES



Definição:

O <u>puxador</u> ("handle") de uma forma sentencial é sua frase simples mais a esquerda.

Definição:

Uma derivação *xUy => xuy* é dita <u>derivação canônica</u> se *y* contém apenas símbolos terminais. Notação xUy ≠> xuy.

Definição:

Temos uma <u>seqüência de derivação canônica</u>. *w =>+ v*, se todas as suas derivações diretas são canônicas. Notação: w ≠>+ v.

<u>Análise de Gramáticas de Precedência</u>

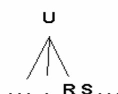
Dada uma forma sentencial *w*, como podemos descobrir quem é o puxador?

Olhando, da esquerda para a direita, cada dois símbolos adjacentes por vez até encontrar o rabo do puxador. Voltar então, procurando a cabeça do puxador, olhando novamente dois símbolos adjacentes por vez.

Para facilitar essa tarefa, vamos introduzir algumas relações em gramáticas de precedência.

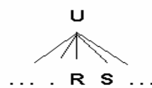
Considerando uma forma sentencial qualquer w = ...RS..., onde R e S são símbolos da gramática e S, R ∈ V, temos três possibilidades:

- **R é um símbolo do puxador, mas S não;**



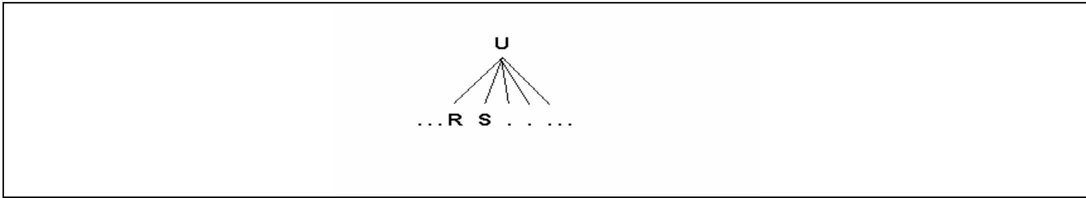
Nós dizemos que R tem precedência sobre S, ou que R é maior que S porque deve ser reduzido primeiro. Notação: R → S. Note que R deve ser o rabo de alguma regra U ::= ...R.

- **R e S são símbolos do mesmo puxador.**



Nós dizemos que R tem a mesma precedência de S e que devem ser reduzidos de uma única vez. Logo, deve existir uma regra U ::= ...RS... . Notação R ± S.

- S é um símbolo do puxador mas R não.



Nós dizemos então que R tem menor precedência que S. Note que S deve ser a cabeça de alguma regra U ::= S... . Notação: R ← S.

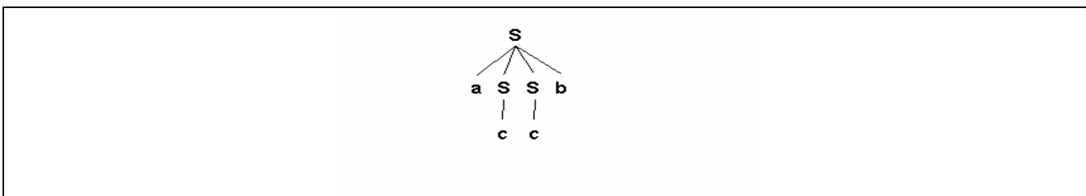
OBSERVAÇÃO:

As relações de precedência apresentadas NÃO são simétricas. Por exemplo, R ← S, não implica que S → R.

Exemplo: Dada a gramática G(S):

S ::= a S S b
S ::= c

vamos tirar as relações que podem ser derivadas das árvores sintáticas apresentadas a seguir.



forma sentencial: a c c b

Puxador: c (o primeiro)

Relações: a ± S, S ± S, S ± b, a ← c, c → c, c → b

```
            S
       a  S  S  b
a S S b        a S S b
```

forma sentencial: a a S S b a S S b b

Puxador: a S S b

Relações: a ← a, b → a, b → b


```
                 S
            a  S  S  b
     a S S b         a S S b
a S S b   c               a S S b
```

forma sentencial: a a a S S b c b a S a S S b b b

Puxador: a S S b

Relações: b → c, S ← a, S ← c

Vamos armazenar essas relações em uma matriz de precedência, onde cada elemento T[i, j] contém a relação associada ao par de símbolos (R_i, S_j).

	S	a	b	c
S	±	←	±	←
a	±	←		←
b	→	→	→	→
c	→	→	→	→

Como a matriz de precedência pode nos ajudar?

Se para cada par de símbolos (R, S) existe no máximo uma relação de precedência, estão através dessas relações nós podemos identificar quem é o puxador de uma dada forma sentencial.

Definição:

Seja $w = S_1S_2...S_N$ uma forma sentencial. O <u>puxador</u> de w é a subcadeia $S_iS_{i+1}...S_j$ mais à esquerda tal que:

$$S_{i-1} \leftarrow S_i \pm S_{i+1} \pm ... \pm S_j \rightarrow S_{j+1}$$

Vamos utilizar essa idéia para verificar se a sentença w = aacbab pertence ou não à linguagem gerada pela gramática para a qual construímos a matriz de precedência.

PASSO	FORMA SENTENCIAL	PUXADOR	REDUÇÃO DO PUXADOR	DERIVAÇÃO DIRETA CONSTRUIDA
1	a ← a ← c → c → b → c → b	c	S	aaScbcb => aaccbcb
2	a ← a ± S ← c → b → c → b	c	S	aaSSbcb => aaScbcb
3	a ← a ± S ± S ± b → c → b	a S S b	S	aScb => aaSSbcb
4	a ± S ← c → b	c	S	aSSb => aScb
5	a ± S ± S ± b	a S S b	S	S => aSSb

Nós definimos as três relações de precedência em termos de árvores de derivação de algumas formas sentenciais. Vamos agora, redefini-las em função das produções da gramática.

Definição:

Dada uma gramática G, as relações de precedência entre seus símbolos são:

R ± S se existe uma produção U ::= ...RS... em G;

R ← S se existe uma produção U ::= ...RV... em G e V =>+ S... ;

R → S sendo S um símbolo terminal, se existe uma produção U ::= ...VW... e V =>+ ...R e W =>* S... .

Definição:

Uma gramática G é chamada de precedência simples se:

1) Entre dois símbolos seus existe no máximo uma relação de precedência;

2) Todas as produções de G tiverem lados direitos únicos.

OBSERVAÇÃO:

Durante a análise, consideramos que cada forma sentencial aparece entre os símbolos '$' (assumindo que '$' não é um símbolo da gramática). Além disso, convencionamos que $ ← R e R → $ para qualquer símbolo R da gramática.

IMPLEMENTAÇÃO DO ANALISADOR ASCENDENTE DE PRECEDÊNCIA SIMPLES

Estrutura de dados

- uma tabela (matriz) de precedência com valores:

  T[i, j] = 0 se não existe relação entre S_i e S_j
  T[i, j] = 1 se S_i ← S_j
  T[i, j] = 2 se S_i ± S_j
  T[i, j] = 3 se S_i → S_j

- uma tabela que armazene as produções de tal forma que dado um lado direito, possamos localizá-lo na tabela e identificar o correspondente lado esquerdo;

- uma pilha onde são armazenados os símbolos da seqüência de entrada (processados da esquerda para a direita) até que seja identificada uma relação → entre o símbolo do topo da pilha e o símbolo de entrada. Isto significa que o símbolo do topo da pilha é o rabo de um puxador. O puxador deve ser identificado e reduzido para o não-terminal correspondente. O processo é repetido até que a pilha só contenha o símbolo inicial da gramática e o próximo símbolo de entrada seja '$'.

Algoritmo do Analisador Ascendente de Precedência Simples

```
início
    pilha[ 1 ] = $; topo = 1;

    leia_próximo_símbolo();

    repita {
        enqto pilha[topo] ¬→ próximo_símbolo {
            empilha( próximo_símbolo );
            leia_próximo_símbolo();
        }

        j = topo;
        enquanto pilha[ j - 1 ] ¬← pilha[ j ] faça
            j = j - 1;

        se existe produção com lado direito =
          pilha[ j ]...pilha[ topo ] então {
            topo = j;
            pilha[ topo] = lado_esquerdo_da_produção;
        }
    } até que não seja possível efetuar redução;

    se topo = 2 e pilha[ topo ] = símbolo_inicial e
      próximo_símbolo = $ então
        SUCESSO();
    senão
        ERRO();
fim
```


Vejamos novamente a análise da sentença w = aaccbcbc, agora com o algoritmo dado.

PASSO	PILHA	RELAÇÃO	SÍMBOLO DE ENTRADA	CADEIA NÃO RECONHECIDA
0	$	←	a	a c c b c b $
1	$ a	←	a	c c b c b $
2	$ a a	←	c	c b c b $
3	$ a a c	→	c	b c b $
4	$ a a S	←	c	b c b $
5	$ a a S c	→	b	c b $
6	$ a a S S	±	b	c b $
7	$ a a S S b	→	c	b $
8	$ a S	←	c	b $
9	$ a S c	→	b	$
10	$ a S S	±	b	$
11	$ a S S b	→	$	
12	$ S	→	$	