

Tema 4

Análise Sintáctica

Gramáticas independentes de contexto, análise sintáctica descendente e ascendente

Compiladores, 2º semestre 2018-2019

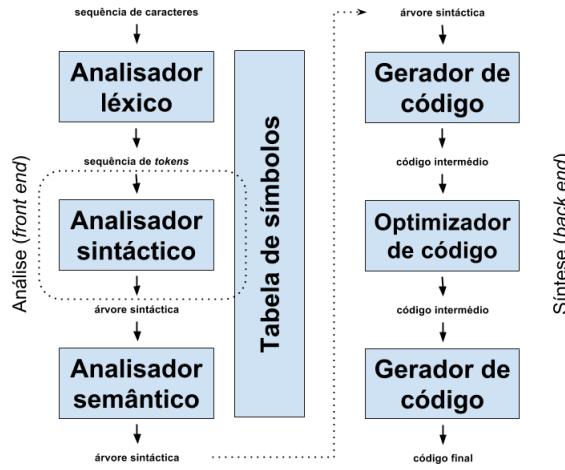
Miguel Oliveira e Silva, Artur Pereira, DETI, Universidade de Aveiro

Conteúdo

1 Análise sintáctica: Estrutura de um Compilador	2
2 Gramáticas Independentes de Contexto	2
2.1 Derivações	3
2.2 Árvore sintáctica	5
2.3 Gramáticas ambíguas	6
2.4 Projecto de gramáticas simples	7
2.5 Operações sobre GIC	8
2.5.1 Reunião	8
2.5.2 Concatenação	9
2.5.3 Fecho de Kleene	10
3 Definição de gramáticas	10
3.1 Símbolos inacessíveis	10
3.2 Critérios de sanidade em gramáticas	11
4 Transformações de gramáticas	12
4.1 Eliminação de produções- ϵ	13
4.2 Eliminação de recursividade à esquerda	13
4.3 Factorização à esquerda	14
4.4 Eliminação de ambiguidades	15
5 Análise sintáctica descendente	17
5.1 Os conjuntos de análise <i>first</i> , <i>follow</i> e <i>lookahead</i>	18
5.2 Análise sintáctica descendente recursiva	19
5.3 Análise sintáctica descendente preditiva	19
5.4 Análise sintáctica descendente preditiva não recursiva	21
5.5 Análise sintáctica descendente em ANTLR4	21
6 Análise sintáctica ascendente	23
6.1 Redução	23
6.2 Análise sintáctica por deslocamento e redução	23
6.3 Conflitos	24
6.4 Construção de um analisador ascendente	26
6.4.1 Analisador ascendente LR simples (SLR)	26

1 Análise sintáctica: Estrutura de um Compilador

- Vamos agora analisar com mais detalhe a fase de análise sintática:



2 Gramáticas Independentes de Contexto

Definição de gramática

- Uma gramática é um quádruplo $G = (T, N, S, P)$, onde:
 - T é um conjunto finito não vazio de símbolos terminais;
 - N é um conjunto finito não vazio, disjunto de T ($N \cap T = \emptyset$), de símbolos não terminais;
 - $S \in N$ é o símbolo inicial;
 - P é um conjunto finito de produções (ou regras de rescrita), cada uma da forma $\alpha \rightarrow \beta$.
- α e β são designados por *cabeça da produção* e *corpo da produção*, respectivamente.
- No caso geral:

$$\alpha \in (T \cup N)^*$$

$$\beta \in (T \cup N)^*$$

isto é, α é uma cadeia de símbolos terminais e não terminais contendo, pelo menos, um símbolo não terminal; e β é uma cadeia de símbolos terminais e não terminais.

Gramáticas independentes de contexto

- Uma gramática $G = (T, N, S, P)$ diz-se independente ou livre de contexto se, para qualquer produção $(\alpha \rightarrow \beta) \in P$, a condição seguinte é satisfeita:

$$\alpha \in N$$

- A linguagem gerada por uma gramática independente do contexto diz-se independente do contexto.
 - Note que as gramáticas regulares são também gramáticas independentes do contexto.
- As gramáticas independentes do contexto são fechadas sob as operações de reunião, concatenação e fecho.
 - No entanto, as GIC não são fechadas sob as operações de intersecção e complementação.

Exemplo de gramática

- Por exemplo a gramática $G = (\{0,1\}, \{S,A\}, S, P)$, onde P é constituído pelas regras:

$$\begin{aligned} S &\rightarrow 0S \\ S &\rightarrow 0A \\ A &\rightarrow 0A1 \\ A &\rightarrow \epsilon \end{aligned}$$

- A linguagem definida por esta gramática é:

$$L = \{0^n 1^m : n \in \mathbb{N} \wedge m \in \mathbb{N}_0 \wedge n > m\}$$

- Esta gramática é um exemplo de uma *gramática independente (ou livre) de contexto*.
- Dada a gramática $G_a = (\{0,1\}, \{A\}, A, P)$ (para as mesmas produções P), que linguagem será definida por G_a ?

Outro exemplo de gramática

- Vamos supor que queremos definir uma gramática, num alfabeto binário ($B = \{\mathbf{0}, \mathbf{1}\}$), que garanta que o número de uns é igual ao números de zeros. Isto é para a linguagem:
 $L = \{u \in B^* \mid \#(\mathbf{0}, u) = \#(\mathbf{1}, u)\}$
- Para chegar a esta gramática podemos desde logo constatar o seguinte:
 - caso apareça um **0** terá de aparecer um **1**;
 - a ordem dos símbolos é irrelevante (não pode depender de permutações).
- Para garantir isso basta ter um conta as duas variantes:

$$S \rightarrow \epsilon \mid \mathbf{1}S\mathbf{0}S \mid \mathbf{0}S\mathbf{1}S$$

- Esta gramática não é regular já que não é possível encostar os terminais totalmente à esquerda (ou à direita).
- É sim um outro exemplo de uma *gramática independente (ou livre) de contexto*.

2.1 Derivações

- Podemos tratar as produções como regras de rescrita, substituindo um símbolo não terminal pelo corpo de uma das suas produções.
- Por exemplo, se tivermos a seguinte gramática:

$$E \rightarrow E * E \mid E + E \mid -E \mid (E) \mid \mathbf{id}$$

podemos rescrever o símbolo não terminal E por qualquer uma das alternativas aplicáveis. Por exemplo:

$$E \Rightarrow -E$$

que se deve ler: “ E deriva $-E$ ”.

- Uma rescrita mais extensa poderá ser:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

- Chamamos a uma sequência deste género, uma *derivação* de $-(\mathbf{id})$ a partir de E .
- Esta derivação prova que a palavra $-(\mathbf{id})$ é um caso particular de uma expressão E .

Derivações: definição geral

- Dada uma palavra $\alpha A \beta$ e uma produção $A \rightarrow v$, chama-se *derivação direta* à rescrita de $\alpha A \beta$ em $\alpha v \beta$, denotando-se

$$\alpha A \beta \Rightarrow \alpha v \beta$$

- Dada uma palavra $\alpha A \beta$, com $\beta \in T^*$, e uma produção $A \rightarrow v$, chama-se *derivação direta à direita* à rescrita de $\alpha A \beta$ em $\alpha v \beta$, denotando-se

$$\alpha A \beta \xrightarrow{D} \alpha v \beta$$

- Dada uma palavra $\alpha A \beta$, com $\alpha \in T^*$, e uma produção $A \rightarrow v$, chama-se *derivação direta à esquerda* à rescrita de $\alpha A \beta$ em $\alpha v \beta$, denotando-se

$$\alpha A \beta \xrightarrow{E} \alpha v \beta$$

- Isto é, é possível fazer uma derivação directa à direita/esquerda quando o lado direito da produção tem terminais à direita/esquerda da regra a rescrever.
- Chama-se *derivação* a uma sucessão de zero ou mais derivações diretas, denotando-se

$$\alpha \xrightarrow{*} \beta$$

- Chama-se *derivação à direita* a uma sucessão de zero ou mais derivações diretas à direita, denotando-se

$$\alpha \xrightarrow{D^*} \beta$$

ou, equivalentemente,

$$\alpha = \alpha_0 \xrightarrow{D} \alpha_1 \xrightarrow{D} \dots \xrightarrow{D} \alpha_n = \beta$$

onde n é o comprimento da derivação.

- Chama-se *derivação à esquerda* a uma sucessão de zero ou mais derivações diretas à esquerda, denotando-se

$$\alpha \xrightarrow{E^*} \beta$$

ou, equivalentemente,

$$\alpha = \alpha_0 \xrightarrow{E} \alpha_1 \xrightarrow{E} \dots \xrightarrow{E} \alpha_n = \beta$$

onde n é o comprimento da derivação.

- Recuperando a gramática:

$$E \rightarrow E * E \mid E + E \mid -E \mid (E) \mid \text{id}$$

- Podemos verificar que a palavra $-(\text{id} + \text{id})$ é um caso particular desta gramática, porque existe (pelo menos) uma derivação:

$$E \xrightarrow{E} -E \xrightarrow{E} -(E) \xrightarrow{E} -(E+E) \xrightarrow{E} -(\text{id} + E) \xrightarrow{E} -(\text{id} + \text{id})$$

- Isto é: $E \xrightarrow{*} -(\text{id} + \text{id})$
- Em cada passo da derivação há duas opções a tomar: escolher qual o não terminal a substituir e, tendo feito esta escolha, escolher a produção desse não terminal.
- A palavra $-(\text{id} + \text{id})$ tem uma derivação alternativa (que difere da anterior na penúltima rescrita) dada por:

$$E \xrightarrow{D} -E \xrightarrow{D} -(E) \xrightarrow{D} -(E+E) \xrightarrow{D} -(E+\text{id}) \xrightarrow{D} -(\text{id} + \text{id})$$

Exemplo de derivação

- Considere, sobre o alfabeto $T = \{a, b, c\}$, a gramática seguinte

$$\begin{aligned} S &\rightarrow \varepsilon \mid aB \mid bA \mid cS \\ A &\rightarrow aS \mid bAA \mid cA \\ B &\rightarrow aBB \mid bS \mid cB \end{aligned}$$

- Determine as derivações à direita e à esquerda da palavra **aabcbc**

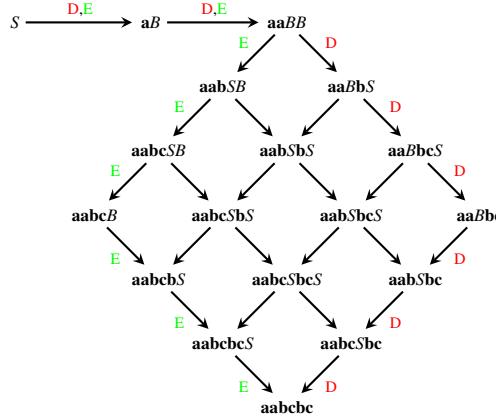
- à direita:

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbcS \\ &\Rightarrow aaBbc \Rightarrow aabSbc \Rightarrow aabcSbc \Rightarrow aabcbc \end{aligned}$$

- à esquerda:

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aabSB \Rightarrow aabcSB \\ &\Rightarrow aabcB \Rightarrow aabcbS \Rightarrow aabcbs \Rightarrow aabcbc \end{aligned}$$

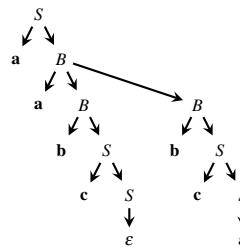
- A árvore seguinte capta as alternativas de derivação. Considera-se novamente a palavra **aabcbc** e a gramática anterior.



- Note que esta árvore contém todas as combinações possíveis de derivações à esquerda e à direita.

2.2 Árvore sintáctica

- Uma *árvore sintáctica*, ou árvore de derivação (*parse tree*), é uma representação de uma derivação onde os nós-ramos são elementos de N (não terminais) e os nós-folhas são elementos de T (terminais).
- A árvore sintáctica da palavra **aabcbc** na gramática anterior é:



- A relação entre gramáticas independentes de contexto e a estrutura de dados tipo árvore é mais profunda do que pode parecer à primeira vista.
- O facto de nas GICs todas as produções terem um, e um só, símbolo não terminal na cabeça da produção, faz com que este possa ser reescrito (derivado) em função da sequência de símbolos do corpo (aplicável) da produção.

- Ou seja, numa estrutura de dados tipo árvore em que a cabeça da produção é o nó “pai”, e a sequência de símbolos do corpo da produção são os filhos.
- O facto da decisão sobre possíveis derivações depender apenas de um (e um só) símbolo não terminal, mais torna eficiente algoritmos automáticos para esse reconhecimento.
- Nas gramáticas que não são independentes de contexto (dependentes de contexto ou sem restrição) esta representação não é possível.

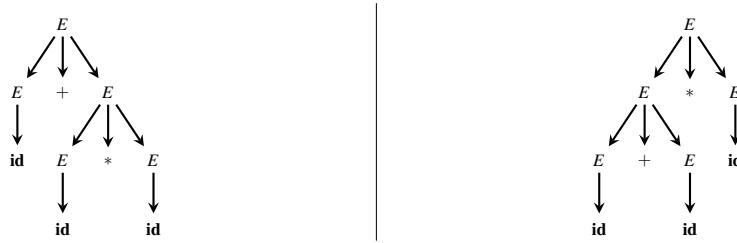
2.3 Gramáticas ambíguas

- Recuperando novamente a gramática:

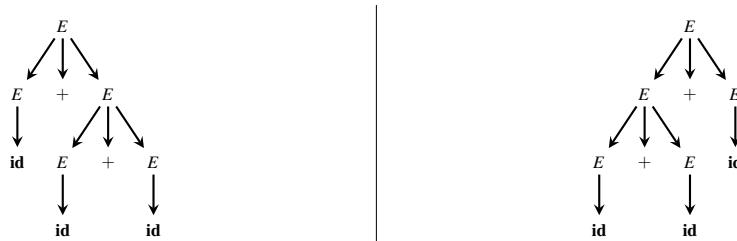
$$E \rightarrow E * E \mid E + E \mid -E \mid (E) \mid \text{id}$$

considere a entrada **id + id * id**

- Podemos extraír duas árvores sintácticas distintas:



- A árvore da esquerda segue as regras de precedência da aritmética, e a da direita dá maior prioridade à soma.
- Este problema não resulta apenas da precedência entre duas operações distintas, a associatividade aplicável à mesma operação pode também gerar ambiguidade.
- Da entrada **id + id + id** também resultam duas árvores sintácticas distintas:



- A árvore da esquerda tem associatividade à direita, e a da direita tem associatividade à esquerda.

Gramáticas ambíguas: definição

- Diz-se que uma palavra é derivada *ambiguamente* se possuir duas ou mais árvores de derivação distintas.
- Diz-se que uma gramática é *ambígua* se possuir pelo menos uma palavra gerada ambiguamente.
- Muitas vezes é possível definir-se uma gramática não ambígua que gere a mesma linguagem do que outra gramática ambígua.
- No entanto, há gramáticas *inherentemente ambíguas*.
- Já vimos um exemplo comum resultante das expressões aritméticas.
- Vejamos outro exemplo:

$$L = \{a^i b^j c^k : i = j \vee j = k\}$$

Gramáticas ambíguas: exemplo 1

- Uma gramática (GIC) para esta linguagem será $G = (\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}, \{S, AB, C, A, BC\}, S, P)$, em que as produções (P) são:

$$\begin{array}{lcl} S & \rightarrow & ABC \mid A BC \\ AB & \rightarrow & \mathbf{a} AB \mathbf{b} \mid \epsilon \\ BC & \rightarrow & \mathbf{b} BC \mathbf{c} \mid \epsilon \\ A & \rightarrow & \mathbf{a} A \mid \epsilon \\ C & \rightarrow & \mathbf{c} C \mid \epsilon \end{array}$$

- Assim, caso $i = j = k$, teremos duas possíveis árvores sintácticas (uma derivando por AB outra por BC).

Gramáticas ambíguas: exemplo 2

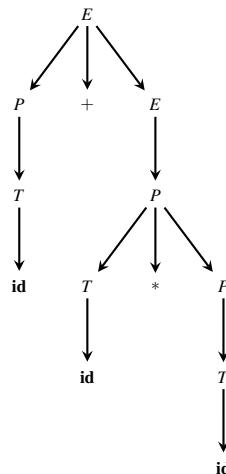
- Recuperando novamente a gramática:

$$E \rightarrow E * E \mid E + E \mid -E \mid (E) \mid \mathbf{id}$$

- Se desejarmos garantir que a operação da multiplicação tem precedência relativamente à soma, sem utilizar convenções que não estejam expressas directamente na gramática (como acontece com a regra ANTLR de dar prioridade às regras definidas primeiro), será que conseguimos gerar uma gramática não ambígua?
- Vejamos se as seguintes regras funcionam:

$$\begin{array}{lcl} E & \rightarrow & P + E \mid P \\ P & \rightarrow & T * P \mid T \\ T & \rightarrow & -E \mid (E) \mid \mathbf{id} \end{array}$$

- Para a entrada $\mathbf{id} + \mathbf{id} * \mathbf{id}$, teremos apenas a seguinte árvore sintáctica:



2.4 Projecto de gramáticas simples

Exemplo de projeto de GIC: solução 1

- Sobre o conjunto de terminais $T = \{\mathbf{a}, \mathbf{b}\}$, determine uma gramática independente do contexto que represente a linguagem:

$$L1 = \{w \in T^* : \#(a, w) = \#(b, w)\}$$

- Anteriormente já foi resolvido um problema similar.

- Logo:

$$S \rightarrow \epsilon \mid \mathbf{a}S\mathbf{b}S \mid \mathbf{b}S\mathbf{a}S$$

- A gramática é ambígua?
- Sim! (Analise a palavra **aabbab**)

Exemplo de projecto de GIC: outras soluções

- Outra possível solução é:

$$\begin{aligned} S &\rightarrow \epsilon \mid \mathbf{a}B \mid \mathbf{b}A \\ A &\rightarrow \mathbf{a}S \mid \mathbf{b}AA \\ B &\rightarrow \mathbf{a}BB \mid \mathbf{b}S \end{aligned}$$

- Ainda outra possibilidade é:

$$\begin{aligned} S &\rightarrow \epsilon \mid \mathbf{a}BS \mid \mathbf{b}AS \\ A &\rightarrow \mathbf{a} \mid \mathbf{b}AA \\ B &\rightarrow \mathbf{a}BB \mid \mathbf{b} \end{aligned}$$

Outro Exemplo de projecto de GIC

- Sobre o conjunto de terminais $T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, determine uma gramática independente do contexto que represente a linguagem:

$$L = \{w \in T^* : \#(a, w) = \#(b, w)\}$$

- Este problema difere do anterior pelo facto de haver mais um símbolo terminal.
- Logo:

$$S \rightarrow \epsilon \mid \mathbf{a}S\mathbf{b}S \mid \mathbf{b}S\mathbf{a}S \mid \mathbf{c}S$$

2.5 Operações sobre GIC

2.5.1 Reunião

- Sejam $G_1 = (T_1, N_1, S_1, P_1)$ e $G_2 = (T_2, N_2, S_2, P_2)$ duas gramáticas independentes de contexto quaisquer com $N_1 \cap N_2 = \emptyset$
- A gramática $G = (T, N, S, P)$ onde:

$$T = T_1 \cup T_2$$

$$N = N_1 \cup N_2 \cup \{S\}$$

$$S \notin (T \cup N_1 \cup N_2)$$

$$P = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2$$

é independente de contexto e gera a linguagem $L = L(G_1) \cup L(G_2)$

- As produções de G_1 e G_2 são transferidas para a nova gramática sem nenhuma alteração substancial (eventualmente pode ser necessário mudar nomes de símbolos não terminais).
- A nova produção $S \rightarrow S_i$, com $i = 1, 2$, permite que G gere a linguagem $L(G_i)$.

Operações sobre GIC: reunião (exemplo)

- Sobre o conjunto de terminais $T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, determine uma gramática independente de contexto que represente a linguagem

$$L = \{w \in T^* : L_1 \cup L_2\}$$

sabendo que:

$$L_1 = \{w \in T^* : \#(a, w) = \#(b, w)\}$$

$$L_2 = \{w \in T^* : \#(a, w) = \#(c, w)\}$$

- Primeiro vamos gerar as gramáticas para cada linguagem:

$$L_1 : S_1 \rightarrow \epsilon \mid \mathbf{a}S_1\mathbf{b}S_1 \mid \mathbf{b}S_1\mathbf{a}S_1 \mid \mathbf{c}S_1$$

$$L_2 : S_2 \rightarrow \epsilon \mid \mathbf{a}S_2\mathbf{c}S_2 \mid \mathbf{c}S_2\mathbf{a}S_2 \mid \mathbf{b}S_2$$

- Donde resulta a seguinte solução:

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow \epsilon \mid \mathbf{a}S_1\mathbf{b}S_1 \mid \mathbf{b}S_1\mathbf{a}S_1 \mid \mathbf{c}S_1 \\ S_2 &\rightarrow \epsilon \mid \mathbf{a}S_2\mathbf{c}S_2 \mid \mathbf{c}S_2\mathbf{a}S_2 \mid \mathbf{b}S_2 \end{aligned}$$

2.5.2 Concatenação

- Sejam $G_1 = (T_1, N_1, S_1, P_1)$ e $G_2 = (T_2, N_2, S_2, P_2)$ duas gramáticas independentes de contexto quaisquer com $N_1 \cap N_2 = \emptyset$
- A gramática $G = (T, N, S, P)$ onde:

$$T = T_1 \cup T_2$$

$$N = N_1 \cup N_2 \cup \{S\}$$

$$S \notin (T \cup N_1 \cup N_2)$$

$$P = \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2$$

é independentes de contexto e gera a linguagem $L = L(G_1) \cdot L(G_2)$

- A nova produção $S \rightarrow S_1 S_2$ justapõe palavras de $L(G_2)$ às de $L(G_1)$.

Operações sobre GIC: concatenação (exemplo)

- Sobre o conjunto de terminais $T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, determine uma gramática independente de contexto que represente a linguagem

$$L = L_1 \cdot L_2$$

sabendo que:

$$L_1 = \{w_1 \in T^* : \#(a, w_1) = \#(b, w_1)\}$$

$$L_2 = \{w_2 \in T^* : \#(a, w_2) = \#(c, w_2)\}$$

- Solução possível:

$$\begin{aligned} S &\rightarrow S_1 S_2 \\ S_1 &\rightarrow \epsilon \mid \mathbf{a}S_1\mathbf{b}S_1 \mid \mathbf{b}S_1\mathbf{a}S_1 \mid \mathbf{c}S_1 \\ S_2 &\rightarrow \epsilon \mid \mathbf{a}S_2\mathbf{c}S_2 \mid \mathbf{c}S_2\mathbf{a}S_2 \mid \mathbf{b}S_2 \end{aligned}$$

2.5.3 Fecho de Kleene

- Seja $G_1 = (T_1, N_1, S_1, P_1)$ uma gramática independente de contexto qualquer.
- A gramática $G = (T, N, S, P)$ onde:

$$T = T_1$$

$$N = N_1 \cup \{S\}$$

$$S \notin (T \cup N_1)$$

$$P = \{S \rightarrow \epsilon, S \rightarrow S_1 S\} \cup P_1$$

é independente de contexto e gera a linguagem $L = (L(G_1))^*$

Operações sobre GIC: fecho de Kleene (exemplo)

- Sobre o conjunto de terminais $T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, determine uma gramática independente de contexto que represente a linguagem

$$L = L_1^*$$

sabendo que:

$$L_1 = \{w \in T^* : \#(a, w) = \#(b, w)\}$$

- Solução possível:

$$\begin{aligned} S &\rightarrow \epsilon \mid S_1 S \\ S_1 &\rightarrow \epsilon \mid \mathbf{a}S_1 \mathbf{b}S_1 \mid \mathbf{b}S_1 \mathbf{a}S_1 \mid \mathbf{c}S_1 \end{aligned}$$

3 Definição de gramáticas

3.1 Símbolos inacessíveis

- Tal como na construção de um programa é trivial definir funções que nunca são utilizadas no programa, também nas gramáticas podemos ter regras inacessíveis.
- Seja $G = (T, N, S, P)$ uma gramática qualquer. Um símbolo X diz-se *acessível* se for possível transformar o símbolo inicial (S) numa expressão que contenha esse símbolo. Ou seja,

$$S \xrightarrow{*} \alpha X \beta \quad \text{com} \quad \alpha, \beta \in (N \cup T)^*$$

- Caso contrário, o símbolo não terminal diz-se *inacessível*.

Símbolos inacessíveis: exemplo

- Considere a gramática $G = (\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}, \{S, C, D, X\}, S, P)$, com as produções P :

$$\begin{aligned} S &\rightarrow \epsilon \mid \mathbf{a}S\mathbf{b} \mid \mathbf{c}C\mathbf{c} \\ C &\rightarrow \mathbf{c}S\mathbf{c} \\ D &\rightarrow \mathbf{d}X\mathbf{d} \\ X &\rightarrow CC \end{aligned}$$

- É impossível transformar S numa expressão que contenha D , \mathbf{d} ou X , pelo que esses símbolos são inacessíveis.
- Os restantes símbolos são acessíveis.

Eliminação de símbolos inacessíveis

- Podemos definir um algoritmo para descobrir os símbolos acessíveis (logo, que permita a eliminação dos restantes).
- O conjunto de símbolos acessíveis V_A pode ser determinado com o seguinte algoritmo:

```

 $V_A = \{S\}$ 
 $N_X = V_A$ 
while  $N_X \neq \emptyset$  do
     $A = \text{element-of}(N_X)$ 
     $N_X = N_X \setminus \{A\}$ 
    foreach  $(A \rightarrow \alpha) \in P$  do
        foreach  $x \in \alpha$  do
            if  $x \notin V_A$  then
                 $V_A = V_A \cup \{x\}$ 
            if  $x \in N$  then
                 $N_X = N_X \cup \{x\}$ 
            end//if
        end//if
    end//foreach
end//foreach
end//while

```

3.2 Critérios de sanidade em gramáticas

Recursividade: Condições de Sanidade

- Por agora deve começar a ser bem evidente duas características essenciais das gramáticas:
 1. são expressas de forma *declarativa*;
 2. são muitas vezes, directa ou indirectamente, *recursivas*.
- Como exemplo, a gramática seguinte tem regras recursivas (directa e indirectamente):

$$\begin{aligned}
 \text{instruction} &\rightarrow \varepsilon \mid \text{conditional} \mid \dots \\
 \text{conditional} &\rightarrow \text{if expression then instruction} \mid \\
 &\quad \text{if expression then instruction else instruction} \\
 \text{expression} &\rightarrow \text{expression } '+' \text{ expression} \mid \\
 &\quad \dots \\
 &\quad \dots
 \end{aligned}$$

- A regra *expression* é directamente recursiva, e a regra *instruction* é indirectamente recursiva.
- Não é assim de estranhar que deva ser recuperada para o campo das gramáticas as *condições de sanidade* aplicáveis às funções recursivas (independentemente das gramáticas serem implementadas por funções).
- Uma definição recursiva útil requer que:
 1. Exista pelo menos uma alternativa não recursiva (**CASO(S) LIMITE**);
 2. Todas as alternativas recursivas ocorram num contexto diferente do original (**VARIABILIDADE**);
 3. Para todas as alternativas recursivas, a mudança do contexto (2) levam-nas mais próximo de, pelo menos, uma alternativa não recursiva (1) (**CONVERGÊNCIA**).
- As condições (1) e (2) são *necessárias*. As três juntas são *suficientes* para garantir a terminação da recursão.
- A aplicação destes conceitos às gramáticas, permite que se definam condições de sanidade a elas aplicadas.

- Como resultado destas condições, para garantir que uma qualquer *gramática* $G = (T, N, S, P)$ faça sentido, todos os seus símbolos não terminais (Z) têm de poder ser derivados numa sequência de símbolos terminais.
- Isto é: $Z \xrightarrow{*} u \wedge u \in T^*$ com $Z \in N$

1. CASO(S) LIMITE:

- Para cumprir esta condição basta garantir que cada símbolo não terminal tenha sempre directa ou indirectamente uma alternativa não recursiva.

2. VARIABILIDADE:

- Para esta condição é necessário que a regra não tenha nenhum alternativa em que esteja definida (directa ou indirectamente) apenas à custa de si própria.
- Isto é, só podemos voltar à mesma regra depois de existir pelo menos uma derivação diferente da regra.

3. CONVERGÊNCIA:

- Às condições anteriores acresce a condição de só podermos voltar à mesma regra depois de existir pelo menos uma derivação que garanta o consumo (antes ou depois) de pelo menos um símbolo terminal.
- Para garantir que a regra como um todo faça sentido, esta condição tem de ser aplicável a todas as definições alternativas da regra.
- Uma gramática fará sentido, caso estas condições se apliquem ao símbolo inicial.
- Como exemplo, considere a seguinte gramática $G = (\{a, b, c\}, \{S, U, V, W, X\}, S, P)$, com P :

$$\begin{array}{lcl} S & \rightarrow & aSb \mid U \mid V \mid W \\ U & \rightarrow & c \\ V & \rightarrow & aV \\ W & \rightarrow & X \\ X & \rightarrow & X \end{array}$$

- Requisito 1
 - as regras $\{S, U\}$ cumprem;
 - as regras $\{V, W, X\}$ não cumprem.
- Requisito 2
 - as regras $\{S, U, V, W\}$ cumprem;
 - a regra $\{X\}$ não cumpre.
- Requisito 3
 - a regra $\{U\}$ cumpre;
 - nenhuma das restantes cumpre.
- Como o símbolo inicial (S) não cumpre as três condições, não há a garantia de que esta gramática faça sentido.

4 Transformações de gramáticas

- Em geral, existem inúmeras gramáticas capazes de reconhecer a mesma linguagem.
- Duas gramáticas dizem-se *equivalentes* se reconhecerem exactamente a mesma linguagem.
- Assim sendo, é possível modificar gramáticas sem alterar a linguagem reconhecida.
- Estas transformações podem ser justificadas pelas seguintes razões:
 - Simplificação da gramática;

- Eliminar ambiguidades da gramática;
 - Reduzir a redundância da gramática;
 - Facilitar a interpretação ou a geração de código;
 - Permitir ou facilitar a sua implementação automática.
- Já vimos anteriormente algoritmos para transformar gramáticas por forma a eliminar símbolos inatingíveis.
 - Vamos agora analisar outras transformações.

4.1 Eliminação de produções- ϵ

- Uma produção- ϵ é uma produção do tipo $Z \rightarrow \epsilon$, para um qualquer símbolo não terminal Z .
- Se tivermos uma linguagem independente de contexto que não contém a palavra vazia (i.e. que não a aceita), então é sempre possível transformar a sua gramática numa gramática equivalente sem este tipo de produções.
- Considere a gramática:

$$\begin{array}{l} S \rightarrow 0S \mid 1U \\ U \rightarrow \epsilon \mid 0U \mid 1S \end{array}$$

que descreve a linguagem formada pelas palavras definidas sobre o alfabeto $\{0, 1\}$, com um número ímpar de 1s.

- A existência da produção- ϵ faz com que as produções $S \rightarrow 1U$ e $U \rightarrow 0U$ possam produzir as derivações $S \Rightarrow 1$ e $U \Rightarrow 0$.
- Assim, podemos eliminar esta produção acrescentando as produções $S \rightarrow 1$ e $U \rightarrow 0$:

$$\begin{array}{l} S \rightarrow 0S \mid 1U \mid 1 \\ U \rightarrow 0U \mid 1S \mid 0 \end{array}$$

- Em geral, o papel da produção $Z \rightarrow \epsilon$ sobre uma produção $U \rightarrow \alpha Z \beta$ pode ser substituído pela inclusão, como alternativa, da produção $U \rightarrow \alpha \beta$

4.2 Eliminação de recursividade à esquerda

- Diz-se que uma gramática é *recursiva à esquerda* se possuir um símbolo não terminal Z que admite uma derivação do tipo $Z \stackrel{*}{\Rightarrow} Z\beta$
- Isto é, se for possível, em um ou mais passos de derivação, transformar uma expressão Z numa expressão que tem Z no início.
- A gramática seguinte é recursiva à esquerda:

$$\begin{array}{l} S \rightarrow UV \\ U \rightarrow \epsilon \mid S+ \\ V \rightarrow a \mid b \mid (S) \end{array}$$

- A derivação $S \Rightarrow UV \Rightarrow S+V$ mostra que é possível transformar S numa expressão com S à esquerda.
- Logo esta gramática tem recursividade à esquerda associada ao símbolo não terminal S .
- Se a recursividade à esquerda se faz com apenas um passo na derivação, então diz-se que a recursividade é *imediata* (ou directa).
- Esta situação só ocorre se a gramática possuir uma ou mais produções do tipo $Z \rightarrow Z\beta$.
- Na gramática seguinte a recursividade é imediata:

$$\begin{array}{l} E \rightarrow T \mid E + T \\ T \rightarrow a \mid b \mid (E) \end{array}$$

- A eliminação da recursividade à esquerda imediata é simples.

- Para tal basta introduzir um novo símbolo não terminal, e converter a recursividade imediata à esquerda para recursividade à direita.
- Considere a seguinte gramática:

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

- Se observarmos as derivações possíveis a partir de A :

$$A \Rightarrow \beta \quad A \Rightarrow A\alpha \Rightarrow \beta\alpha \quad A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \dots$$

- Constatamos que para n passos ($n > 0$), temos: $A \Rightarrow \beta\alpha^{n-1}$
- Estas palavras podem ser obtidas com a seguinte gramática equivalente:

$$\begin{aligned} A &\rightarrow \beta X \\ X &\rightarrow \epsilon | \alpha X \end{aligned}$$

- Constatamos que se converteu a recursividade à esquerda para uma recursividade à direita.
- Este algoritmo é facilmente generalizável a situações em que há mais do que uma produção com recursividade imediata à esquerda.

$$A \rightarrow \beta_1 | \beta_2 | \dots | \beta_m | A\alpha_1 | A\alpha_2 | \dots | A\alpha_n$$

- Então teremos:

$$\begin{aligned} A &\rightarrow \beta_1 X | \beta_2 X | \dots | \beta_m X \\ X &\rightarrow \epsilon | \alpha_1 X | \alpha_2 X | \dots | \alpha_n X \end{aligned}$$

- O algoritmo pode ser generalizado para eliminar recursividade à esquerda que não seja imediata (se tiver curiosidade, pode ver a bibliografia recomendada).

Eliminação de recursividade à esquerda: exemplo

- Considere a seguinte gramática:

$$\begin{aligned} E &\rightarrow E+T | T \\ T &\rightarrow T^*F | F \\ F &\rightarrow (E) | \mathbf{id} \end{aligned}$$

- Esta gramática tem recursividade à esquerda, aplicando o algoritmo para a transformar numa gramática com recursividade à direita:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' | \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' | \epsilon \\ F &\rightarrow (E) | \mathbf{id} \end{aligned}$$

4.3 Factorização à esquerda

- A factorização à esquerda é uma transformação da gramática que é útil para produzir gramáticas adaptadas a análise sintáctica descendente preditiva.

- Consiste numa operação similar à transformação aritmética de “pôr em evidência” um factor comum.
- No caso, sempre que duas ou mais produções têm o mesmo prefixo (sequência de símbolos terminais e/ou não terminais), então podemos transformar essa gramática por forma a eliminar essa redundância.
- Como exemplo, vamos considerar a seguinte gramática:

$$\begin{aligned} \text{stat} &\rightarrow \text{if expr then stat else stat} \mid \\ &\quad \text{if expr then stat} \end{aligned}$$

- Se observarmos o *token if* não podemos desde logo decidir por qual das duas produções alternativas devemos seguir.
- O problema resolve-se com a seguinte transformação:

$$\begin{aligned} \text{stat} &\rightarrow \text{if expr then stat } U \mid \\ U &\rightarrow \varepsilon \mid \text{else stat} \end{aligned}$$

Factorização à esquerda: algoritmo

- O algoritmo geral para realizar esta transformação é o seguinte:
- Para cada não terminal A , descobrir o maior prefixo α comum a duas ou mais alternativas.
- Se $\alpha \neq \varepsilon$, então a produção:

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$$

- pode ser transformada nas seguintes produções:

$$\begin{aligned} A &\rightarrow \alpha X \mid \gamma \\ X &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

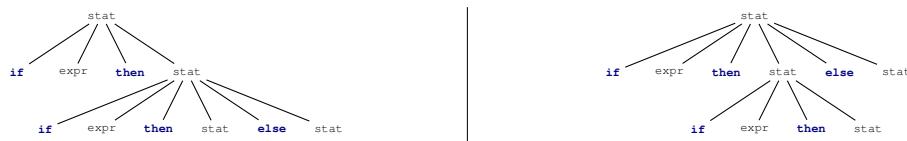
- Esta transformação deve ser repetida até que não se encontrem prefixos comuns.

4.4 Eliminação de ambiguidades

- A gramática seguinte é ambígua.

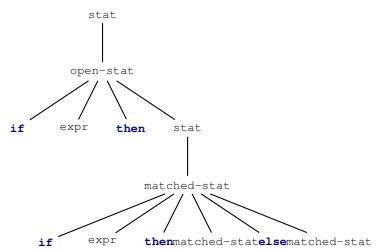
$$\begin{aligned} \text{stat} &\rightarrow \text{if expr then stat} \\ &\mid \text{if expr then stat else stat} \\ &\mid \dots \text{ (outras instruções)} \end{aligned}$$

- Na presença de uma entrada tipo **if if else** não existir duas árvores sintácticas distintas consoante se considera o **else** como sendo do primeiro ou do segundo **if**.



- Podemos reescrever esta gramática seguindo a ideia de que uma instrução (stat) que apareça entre um **then** e um **else** tem de ser terminado:

$$\begin{aligned} \text{stat} &\rightarrow \text{matched-stat} \mid \text{open-stat} \\ \text{matched-stat} &\rightarrow \text{if expr then matched-stat else matched-stat} \\ &\mid \dots \text{ (outras instruções)} \\ \text{open-stat} &\rightarrow \text{if expr then stat} \\ &\mid \text{if expr then matched-stat else open-stat} \end{aligned}$$



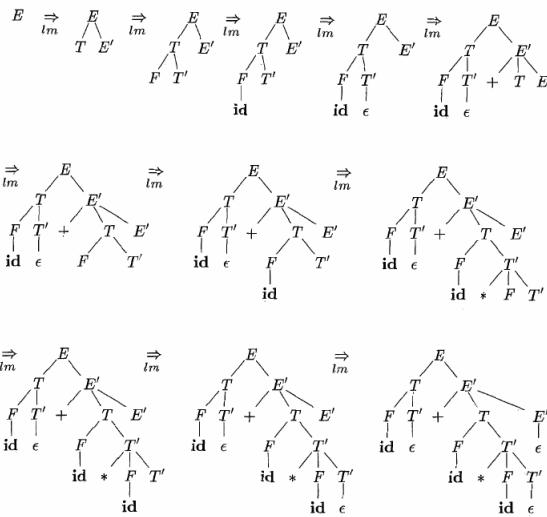
5 Análise sintáctica descendente

- A análise sintáctica descendente pode ser vista como sendo o problema de construir a árvore sintáctica para a sequência de *tokens* de entrada, partindo da raiz (símbolo inicial), e criando os nós da árvore em pré-ordem.
- De forma análoga, podemos ver este tipo de análise como sendo a descoberta da derivação mais à esquerda para a sequência de entrada.
- Recuperando a seguinte gramática:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

- Podemos construir a árvore sintáctica para a entrada: **id+id *id**¹

$$\begin{aligned} E \Rightarrow \textcolor{red}{T}E' \Rightarrow \textcolor{red}{F}T'E' \Rightarrow \textcolor{red}{id}T'E' \Rightarrow \textcolor{red}{id}+\textcolor{red}{T}E' \Rightarrow \textcolor{red}{id}+\textcolor{red}{F}T'E' \Rightarrow \textcolor{red}{id}+\textcolor{red}{id}\textcolor{red}{T}'E' \\ \Rightarrow \textcolor{red}{id}+\textcolor{red}{id}*\textcolor{red}{F}T'E' \Rightarrow \textcolor{red}{id}+\textcolor{red}{id}*\textcolor{red}{id}\textcolor{red}{T}'E' \Rightarrow \textcolor{red}{id}+\textcolor{red}{id}*\textcolor{red}{id}E' \Rightarrow \textcolor{red}{id}+\textcolor{red}{id}*\textcolor{red}{id} \end{aligned}$$



- Existem várias aproximações à análise sintáctica descendente.
- Uma delas – denominada por *análise sintáctica descendente recursiva* – é uma aproximação genérica, mas pode requerer um algoritmo de *backtracking* (tentativa/erro) para descobrir a produção correcta a ser aplicada a cada novo *token* na entrada.
- Outra aproximação mais prática – denominada por *análise sintáctica descendente preditiva* – é um caso especial deste tipo de análise que não requer *backtracking*.
- Vamos estudar três conjuntos de análise – *first*, *follow* e *lookahead* – que simplificam o desenvolvimento destes analisadores.
- A análise sintáctica descendente preditiva escolhe a produção a ser aplicada vendo um número fixo de *tokens* na entrada que ainda não foram consumidos.
- Se o número de *tokens* requerido for apenas um, este tipo de analisadores chamam-se *LL(1)*.
- O primeiro *L* indica que a entrada é analisada da esquerda para a direita; O segundo *L* significa que se escolhe sempre a derivação à esquerda.
- Para *k tokens* os analisadores chamam-se *LL(k)*.
- A construção deste tipo de analisadores pode ser feita *recursivamente por funções* – onde a cada regra não terminal se associa uma função –, ou de uma forma *iterativa através de uma tabela* e explicitando e com uma estrutura de dados tipo pilha.

¹Imagens retiradas do livro: “Compilers: Principles, Techniques, & Tools”, 2ed, Aho, et.al

5.1 Os conjuntos de análise *first*, *follow* e *lookahead*

- Para ajudar na construção de analisadores sintáticos descendentes preditivos, é conveniente a definição das funções – *first* e *follow* – associadas a símbolos da gramática.
- Estas funções calculam e devolvem conjuntos de símbolos terminais.
- Durante o reconhecimento descendente estas funções facilitam (ou mesmo, determinam) a escolha das produções a aplicar tendo em conta os próximos *token* de entrada.
- Para lidar com erros sintáticos, a função *follow* pode ser utilizada como mecanismo de re-sincronização do analisador sintáctico.
- Seja $\alpha \in (T \cup N)^*$ (isto é: uma qualquer sequência de símbolos da gramática), a função $first(\alpha)$ devolve o conjunto de símbolos terminais ($\in T$), que podem começar sequências de *tokens* derivadas de α .
- Se $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, então ε também fará parte desse conjunto.
- Assim um símbolo terminal x pertence a este conjunto se e só se $\alpha \stackrel{*}{\Rightarrow} x\beta$
- A função $follow(A)$, para um símbolo não terminal A , devolve o conjunto de símbolos terminais ($\in T$), que podem aparecer imediatamente a seguir a A (i.e. à sua direita).
- Assim um símbolo terminal x pertence a este conjunto se e só se $S \stackrel{*}{\Rightarrow} \alpha A x \beta$, para quaisquer sequências α e β .
- Chama-se a atenção de que o *token* especial “fim de ficheiro” (EOF ou \$) pode aparecer no conjunto $follow(A)$ se este não terminal for o último da gramática.
- Para calcular a função $first(X)$, para qualquer símbolo X da gramática podemos utilizar o seguinte algoritmo:
 1. Se X é um símbolo terminal, então: $first(X) = \{X\}$.
 2. Se X é um símbolo não terminal e $X \rightarrow Y_1 Y_2 \dots Y_k$ é uma produção, para algum $k \geq 1$, então adicionar x a $first(X)$ se, para algum valor de i , x está em $first(Y_i)$ e se ε está em todos os $first(Y_1), \dots, first(Y_{i-1})$. Isto é, $Y_1 \dots Y_{i-1} \stackrel{*}{\Rightarrow} \varepsilon$. Se ε está em $first(Y_j)$, para $j = 1, 2, \dots, k$, então adicionar ε a $first(X)$.
 3. Se $X \rightarrow \varepsilon$ é uma produção, então adicionar ε a $first(X)$.
- Podemos agora calcular a função *first* para qualquer sequência de símbolos $X_1 X_2 \dots X_n$ de seguinte forma:
 - Adicionar todos os símbolos, excepto ε , de $first(X_1)$ a $first(X_1 X_2 \dots X_n)$.
 - Se ε pertencer a $first(X_1)$, adicionar todos os símbolos, excepto ε , de $first(X_2)$ a $first(X_1 X_2 \dots X_n)$.
 - E assim sucessivamente.
 - Finalmente, adicionar ε a $first(X_1 X_2 \dots X_n)$ se ε pertencer a todos os $first(X_i)$, $i = 1, \dots, n$
- Para calcular a função $follow(X)$, para qualquer símbolo não terminal X da gramática aplicam-se o seguinte algoritmo:
 1. Colocar “fim de ficheiro” em $follow(S)$ (sendo S o símbolo inicial).
 2. Se existir uma produção $A \rightarrow \alpha X \beta$, então adicionar $first(\beta) \setminus \{\varepsilon\}$ a $follow(X)$.
 3. Se existir uma produção $A \rightarrow \alpha X$ ou uma produção $A \rightarrow \alpha X \beta$ onde $\varepsilon \in first(\beta)$, então adicionar $follow(A)$ a $follow(X)$.
- Recuperando novamente a gramática:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \varepsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

temos:

$$\begin{aligned}
first(E) &= first(T) = first(F) = first((E)) \cup first(\text{id}) = \{(), \text{id}\} \\
first(E') &= first(+TE') \cup first(\epsilon) = \{+, \epsilon\} \\
first(T') &= first(*FT') \cup first(\epsilon) = \{*, \epsilon\} \\
follow(E) &= \{\text{EOF}\} \cup \{\} = \{(), \text{EOF}\} \\
follow(E') &= follow(E) \cup follow(E') = follow(E) = \{(), \text{EOF}\} \\
follow(T) &= (first(E') \setminus \{\epsilon\}) \cup follow(E') = \{+, (), \text{EOF}\} \\
follow(T') &= follow(T) \cup follow(T') = follow(T) = \{+, (), \text{EOF}\} \\
follow(F) &= (first(T') \setminus \{\epsilon\}) \cup follow(T') = \{+, *, (), \text{EOF}\}
\end{aligned}$$

O conjunto de análise *lookahead*

- O conjunto *lookahead* (ou de antevisão) aplica-se às produções de uma gramática, e envolve as funções *first* e *follow*.
- É definido por:

$$lookahead(A \rightarrow \alpha) = \begin{cases} first(\alpha) & \text{se } \alpha \text{ não deriva } \epsilon \\ (first(\alpha) \setminus \{\epsilon\}) \cup follow(A) & \text{se } \alpha \xrightarrow{*} \epsilon \end{cases}$$

- Ou seja, é o conjunto de símbolos terminais (sem a palavra vazia, mas eventualmente com o EOF) que iniciam uma produção.

5.2 Análise sintáctica descendente recursiva

- Um analisador deste tipo consiste num conjunto de funções, uma para cada não terminal.
- Para uma produção do tipo $A \rightarrow X_1 X_2 \cdots X_n$ temos a seguinte implementação para a função que lhe está associada:

```

void A() {
    // A → X1 X2 … Xn
    for (i = 1 to n) {
        if (Xi é um símbolo não terminal)
            call function Xi();
        else if (símbolo de entrada reconhecido por Xi)
            avançar a entrada para o próximo símbolo
        else
            erro de reconhecimento
    }
}

```

- Note que com este algoritmo uma produção do tipo $A \rightarrow A \cdots$ gera recursão infinita, razão pela qual não pode haver recursividade à esquerda nas regras neste tipo de analisadores.
- Um analisador descendente recursivo genérico pode requerer *backtracking*, isto é, pode ser necessário andar para trás no consumo de *tokens* de entrada até se descobrir uma derivação correcta.
- No entanto, para a larga maioria das construções de linguagens de programação os analisadores descendentes não requerem *backtracking*.

5.3 Análise sintáctica descendente preditiva

- Podemos construir analisadores descendentes que não requerem *backtracking*.
- São designados por *analisadores descendentes preditivos*.
- Este tipo de analisadores designam-se por $LL(k)$, em que k é o número de *tokens* à frente que o algoritmo requer ($k \geq 1$).
- Em particular, a classe dos analisadores $LL(1)$ é suficientemente expressiva para cobrir a maioria das construções de linguagens de programação.
- No entanto, é necessário algum cuidado na definição das regras, já que os analisadores $LL(1)$ não aceitam gramáticas ambíguas ou com recursividade à esquerda.

- Uma gramática é $LL(1)$ se e só se, para produções do tipo $A \rightarrow \alpha \mid \beta$, as condições seguintes são verdadeiras:
 1. Para qualquer símbolo terminal \mathbf{a} , no máximo, apenas uma das sequências α ou β podem derivar palavras que comecem por \mathbf{a} .
 2. No máximo, apenas uma das sequências α ou β podem derivar ϵ .
 3. Se $\beta \xrightarrow{*} \epsilon$, então α não deriva nenhuma sequência começando com um terminal existente em $follow(A)$. Da mesma forma, se $\alpha \xrightarrow{*} \epsilon$, então β não deriva nenhuma sequência começando com um terminal existente em $follow(A)$.
- Considerando a gramática $G = (T, N, S, P)$, e todos os não-terminais $A \in N$ com produções $P : A \rightarrow \alpha \mid \beta$, as primeiras duas condições são equivalentes à condição:

$$first(\alpha) \cap first(\beta) = \emptyset$$

- A terceira condição, é equivalente a:

$$\epsilon \in first(\beta) \implies first(\alpha) \cap follow(A) = \emptyset$$

(e *vice-versa* trocando β por α)

- Em síntese, uma gramática será $LL(1)$ se um *token* na entrada for sempre suficiente para decidir sem ambiguidade o que fazer.

Análise sintáctica descendente preditiva: tabela de parsing

- Podemos construir uma tabela de *parsing* M para analisadores descendentes preditivos $LL(1)$ que indique as produções a escolher como função do símbolo não-terminal onde estamos e do próximo *token* na entrada.
- Algoritmo: Para cada produção $A \rightarrow \alpha$ da gramática fazer o seguinte:
 1. Para cada símbolo terminal \mathbf{a} existente em $first(A)$, adicionar $A \rightarrow \alpha$ a $M[A, \mathbf{a}]$
 2. Se $\epsilon \in first(A)$, então para cada terminal \mathbf{b} existente em $follow(A)$, adicionar $A \rightarrow \alpha$ a $M[A, \mathbf{b}]$. Se $\epsilon \in first(A) \wedge EOF \in follow(A)$, então adicionar $A \rightarrow \alpha$ a $M[A, EOF]$.

Análise sintáctica descendente preditiva: exemplo

- A gramática apresentada a seguir é $LL(1)$:

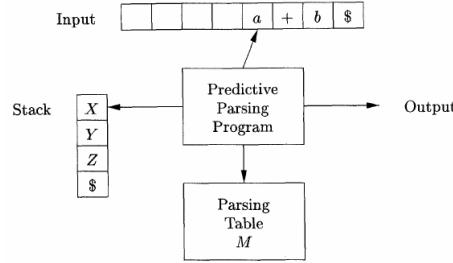
$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \\ E' \rightarrow \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \\ T' \rightarrow \epsilon \\ F \rightarrow (E) \\ F \rightarrow \mathbf{id} \end{array}$$

$$\begin{array}{ll} first(E) & = first(T) = first(F) = \{(\mathbf{,}, \mathbf{id})\} \\ first(E') & = \{+, \epsilon\} \\ first(T') & = \{*, \epsilon\} \\ follow(E) & = follow(E') = \{\mathbf{,}, EOF\} \\ follow(T) & = follow(T') = \{+, \mathbf{,}, EOF\} \\ follow(F) & = \{+, *, \mathbf{,}, EOF\} \end{array}$$

- A tabela de *parsing* associa produções à relação de símbolos não-terminais com os *tokens* de entrada:

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T		$T \rightarrow FT'$			$T \rightarrow FT'$	
T'			$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

5.4 Análise sintáctica descendente preditiva não recursiva



- Podemos assim construir um analisador preditivo sem funções recursivas, implementando-o com uma tabela e uma pilha.
- Para a gramática anterior e a entrada $\text{id}_1 + \text{id}_2 * \text{id}_3$:

<i>reconhecido</i>	<i>pilha</i>	<i>entrada</i>	<i>acção</i>
	$E\$$	$\text{id}_1 + \text{id}_2 * \text{id}_3 \$$	$E \rightarrow TE'$
	$TE' \$$	$\text{id}_1 + \text{id}_2 * \text{id}_3 \$$	$T \rightarrow FT'$
	$FT'E' \$$	$\text{id}_1 + \text{id}_2 * \text{id}_3 \$$	$F \rightarrow id$
	$id T'E' \$$	$\text{id}_1 + \text{id}_2 * \text{id}_3 \$$	reconhece id
id_1	$T'E' \$$	$+ \text{id}_2 * \text{id}_3 \$$	$T' \rightarrow \epsilon$
id_1	$E' \$$	$+ \text{id}_2 * \text{id}_3 \$$	$E' \rightarrow +TE'$
id_1	$+TE' \$$	$+ \text{id}_2 * \text{id}_3 \$$	reconhece +
$\text{id}_1 +$	$TE' \$$	$\text{id}_2 * \text{id}_3 \$$	$T \rightarrow FT'$
$\text{id}_1 +$	$FT'E' \$$	$\text{id}_2 * \text{id}_3 \$$	$F \rightarrow id$
$\text{id}_1 +$	$id T'E' \$$	$\text{id}_2 * \text{id}_3 \$$	reconhece id
$\text{id}_1 + \text{id}_2$	$T'E' \$$	$* \text{id}_3 \$$	$T' \rightarrow *FT'$
$\text{id}_1 + \text{id}_2$	$*FT'E' \$$	$* \text{id}_3 \$$	reconhece *
$\text{id}_1 + \text{id}_2 *$	$FT'E' \$$	$\text{id}_3 \$$	$F \rightarrow id$
$\text{id}_1 + \text{id}_2 *$	$id T'E' \$$	$\text{id}_3 \$$	reconhece id
$\text{id}_1 + \text{id}_2 * \text{id}_3$	$T'E' \$$	$\$$	$T' \rightarrow \epsilon$
$\text{id}_1 + \text{id}_2 * \text{id}_3$	$E' \$$	$\$$	$E' \rightarrow \epsilon$
$\text{id}_1 + \text{id}_2 * \text{id}_3$	$\$$	$\$$	aceita

5.5 Análise sintáctica descendente em ANTLR4

- Como foi já referido por várias vezes, o ANTLR4 é um analisador descendente (em oposição ao yacc/bison que é um analisador ascendente).
- No entanto, a tecnologia de compilação implementada no ANTLR4 tem algumas diferenças relativamente aos analisadores descendentes preditivos apresentados.
- Uma delas, é o facto de ser caracterizado como sendo um analisador preditivo da $LL(*)$ adaptativo.
- Não requer *backtracking*, mas pode observar os *tokens* à frente na entrada até ao fim desta, se necessário for.
- Por outro lado, faz uso de uma análise dinâmica da gramática (em tempo de execução), antes do reconhecimento começar.
- Na prática, isto significa que o ANTLR4 aceita gramáticas bem mais genéricas do que as normalmente associadas aos analisadores $LL(k)$.
- Outra característica de grande interesse prático do ANTLR4, é o facto de aceitar gramáticas com recursividade directa à esquerda.
- Torna-se assim possível definir gramáticas do tipo:

```
grammar Expr;
```

```

main: stat* EOF;
stat: expr;
expr: expr '*' expr # Mult
     | expr '+' expr # Add
     | INT          # Int
     ;
INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;

```

- O que o ANTLR4 faz é automaticamente reescrever as regras com recursividade directa à esquerda em regras equivalentes sem essa característica (utilizando um algoritmo que pode ser similar ao apresentado em 4.2).
- Uma limitação desta característica é facto da recursividade à esquerda ter de ser directa.
- O ANTLR4 tem outras características que o diferenciam, e tornam a sua classificação como analisador descendente um pouco mais complicada.
- É o caso dos predicados semânticos que permitem que a gramática a utilizar possa mudar dinamicamente consoante quaisquer condições expressas na linguagem destino (Java por omissão).
- Esta característica faz com que o ANTLR4 possa, durante a análise sintáctica, implementar construções típicas de gramáticas dependentes de contexto.

6 Análise sintáctica ascendente

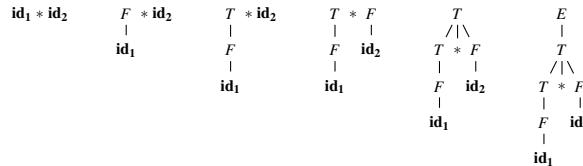
- A análise sintáctica ascendente pode ser vista como sendo o problema de construir a árvore sintática partindo das folhas (*tokens*), e criando os nós da árvore de baixo para cima até à raiz (símbolo inicial).
- Considere a seguinte gramática:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

- Vamos proceder à análise sintáctica processando a entrada **id₁** * **id₂** da esquerda para direita e aplicando de imediato a produção que a vai aproximando do topo.



6.1 Redução

- A análise ascendente pode também ser vista como o processo de *reduzir* a sequência de *tokens* de entrada ao símbolo inicial.
- Em cada passo de redução, a subsequência de símbolos (terminais e/ou não terminais) correspondente ao corpo de uma produção, é substituída pela cabeça da produção.
- A decisão chave neste tipo de análise sintáctica, reside em quando aplicar uma redução e a escolha da produção a aplicar.
- A sequência de reduções aplicadas ao exemplo anterior, gerou a seguinte sequencia de símbolos:

$$\text{id}_1 * \text{id}_2, F * \text{id}_2, T * \text{id}_2, T * F, T, E$$

- Cada elemento desta sequência corresponde à raiz das árvores atrás apresentadas.

Derivação à direita

- Por definição, uma redução é a operação inversa da (respectiva) derivação.
- Assim se reescrevermos a sequência seguindo as derivações (do símbolo inicial até à entrada):

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id}_2 \Rightarrow F * \text{id}_2 \Rightarrow \text{id}_1 * \text{id}_2$$

- Constatamos assim que esta análise corresponde à derivação à direita da sequência de entrada.

6.2 Análise sintáctica por deslocamento e redução

- A análise sintáctica por deslocamento e redução é uma forma de análise ascendente na qual uma pilha vai armazenando símbolos da gramática (por reduzir), e um *buffer* de entrada contém os *tokens* ainda por processar.
- O topo dessa pilha conterá sempre a próxima sequência de símbolos a reduzir.
- Para facilitar a visualização deste processo iremos utilizar o símbolo **\$** para indicar o *token* especial *fim de ficheiro*.
- Podemos assim visualizar o processo de análise ascendente com uma tabela onde se mostra o conteúdo da pilha, o *buffer* de entrada e a acção a tomar.
- Podemos efectuar as seguintes acções:
 1. **redução:** o topo da pilha é substituído pela cabeça da produção (respectiva);

- 2. **deslocamento**: o próximo *token* da entrada é colocado no topo da pilha.
 - 3. **aceitar**: entrada reconhecida com sucesso
 - 4. **rejeição**: erro sintáctico
- Pegando novamente no exemplo dado, teremos a seguinte tabela:

<i>pilha</i>	<i>entrada</i>	<i>acção</i>
\$	id₁ * id₂ \$	deslocamento
\$ id₁	* id₂ \$	redução por $F \rightarrow \text{id}$
\$ F	* id₂ \$	redução por $T \rightarrow F$
\$ T	* id₂ \$	deslocamento
\$ T *	id₂ \$	deslocamento
\$ T * id₂	\$	redução por $F \rightarrow \text{id}$
\$ T * F	\$	redução por $T \rightarrow T * F$
\$ T	\$	redução por $E \rightarrow T$
\$ E	\$	aceitar

- Uma vez que estamos na presença de derivações à direita (embora feitas “ao contrário”, partindo das folhas da árvore para a raiz), é garantido que, a existir, o corpo da produção aparecerá sempre no topo da pilha.
 - Vamos implementar um analisador ascendente por deslocamento/redução para processar expressões aritméticas simples.
 - Para simplificar, vamos considerar que um *token* é um carácter (os números são apenas um dígito).
 - A gramática pretendida é: $e \rightarrow (e) \mid e + e \mid e - e \mid exe \mid e/e \mid D$
- O programa 1 resolve este problema.

6.3 Conflitos

- Existem gramáticas independentes de contexto para as quais a análise sintáctica por deslocamento e redução não funciona.
- Nessas gramáticas, mesmo com o conhecimento do conteúdo actual da pilha e da entrada, o analisador não consegue decidir se deve fazer uma redução ou um deslocamento (conflito do tipo *shift/reduce*), ou não consegue decidir qual das reduções possíveis deve fazer (conflito do tipo *reduce/reduce*).
- Como exemplo deste tipo de situações temos as gramáticas ambíguas.

Conflitos: *shift/reduce*

- Um exemplo de conflitos do tipo *shift/reduce* é a seguinte gramática da instrução condicional:

$$\begin{array}{lcl} \text{stat} & \rightarrow & \text{if expr then stat} \\ & | & \text{if expr then stat else stat} \\ & | & \text{other} \end{array}$$

- Quando o analisador estiver no estado:

<i>pilha</i>	<i>entrada</i>
\$... if expr then stat	else ... \$

pode fazer quer um deslocamento, quer uma redução.

- Uma possibilidade simples para resolver este tipo de conflitos é privilegiar uma dessas acções (por exemplo, o deslocamento).

Listing 1: Exemplo simples de analisador ascendente

```

import static java.lang.System.*;
import java.util.Stack;

public class ExprLR1
{
    public static void main(String[] args) {
        String[] grammar = {"(e)", "e+e", "e-e", "exe", "e:e", "D"};
        Stack<Character> stack = new Stack<>();
        for(int i = 0; i < args.length; i++) {
            if (args[i].length() != 1) {
                err.println("Invalid token \\" + args[i] + "\\");
                exit(1);
            }
            // shift next token:
            char t = args[i].charAt(0);
            if (t >= '0' && t <= '9')
                t = 'D';
            stack.push(t);
            out.printf("[shift]: %5c - stack: %s\n", t, stack);
            boolean reduce;
            // attempt as much reduces as possible:
            do {
                reduce = false;
                for(String p: grammar)
                    if(topMatches(stack,p)) {
                        reduce = true;
                        popN(stack, p.length());
                        stack.push('e');
                        out.printf("[reduce]: %5s - stack: %s\n", p, stack);
                    }
            }
            while(reduce);
        }
        out.println();
        if (stack.size() == 1 && stack.pop() == 'e')
            out.println("Syntactical valid expression!");
        else
            err.println("ERROR: Syntactical invalid expression!");
    }

    static boolean topMatches(Stack<Character> stack, String rule) {
        assert stack != null;
        assert rule != null && !rule.isEmpty();

        boolean res = stack.size() >= rule.length();
        for(int i = 0; res && i < rule.length(); i++)
            res = stack.get(stack.size()-rule.length()+i) == rule.charAt(i);
        return res;
    }

    static void popN(Stack<Character> stack, int n) {
        assert stack != null;
        assert n > 0 && stack.size() >= n;

        for(int i = 0; i < n; i++)
            stack.pop();
    }
}

```

Conflitos: *reduce/reduce*

- Vejamos agora a seguinte gramática:

stat	\rightarrow	id (parameterList)
		id := expr
parameterList	\rightarrow	parameterList, parameter
		parameter
parameter	\rightarrow	id
expr	\rightarrow	id
		id (exprList)
exprList	\rightarrow	exprList, expr
		expr

- Quando o analisador estiver no estado:

<i>pilha</i>	<i>entrada</i>
\$... id (id)	, id ... \$

temos outro tipo de conflito (*reduce/reduce*) já que o analisador não consegue decidir como reduzir o **id** que está no topo da pilha (parameter ou expr?).

6.4 Construção de um analisador ascendente

- O tipo de analisadores ascendentes mais em uso são os chamados *LR(k)*.
- O primeiro *L* indica que a entrada é analisada da esquerda para a direita, e o *R* significa que se escolhe sempre a derivação à direita.
- O *k* é o número de *tokens* de entrada de antevista (*lookahead*).
- Os casos práticos de interesse tem valores de *k* iguais a 0 ou a 1.
- Os analisadores *LR* são baseados em tabelas (à semelhança dos analisadores *LL* não recursivos).
- Este tipo de analisadores sintácticos têm as seguintes características:
 - É possível construir analisadores para virtualmente todo o tipo de linguagens para as quais existem gramáticas independentes de contexto;
 - O método de análise *LR* é o método por deslocamento/redução sem *backtracking* mais genérico conhecido;
 - Permite a detecção de erros sintácticos assim que possível numa análise da entrada sequencial da esquerda para direita;
 - Em teoria, a classe de gramáticas passível de ser analisada supera a dos analisadores *LL* preditivos.
- Uma das suas limitações é ser mais complicado ter acesso ao contexto (*top-down*) envolvido no processo de reconhecimento de uma regra (dificultando a passagem de informação para baixo).
- Outro “problema” é a complexidade no seu desenvolvimento, pelo que requer uma ferramenta para esse fim (*yacc/bison*).

6.4.1 Analisador ascendente *LR* simples (SLR)

- Como é que um analisador por deslocamento/redução decide por uma das acções?
- Por exemplo, na gramática apresentadas com expressões aritméticas, se a pilha contiver $\$T$ e o próximo *token* for $*$, como é que o analisador sabe que o topo da pilha não é para reduzir ($E \rightarrow T$), mas sim deslocar esse símbolo para a pilha?
- Um analisador *SLR* toma estas decisões registando o estado onde está (*item*) no reconhecimento (como um autómato).

- Por exemplo, a produção $A \rightarrow XYZ$ tem 4 itens possíveis (o ponto indica o estado de reconhecimento):

$$\begin{aligned} A &\rightarrow \cdot XYZ \\ A &\rightarrow X \cdot YZ \\ A &\rightarrow XY \cdot Z \\ A &\rightarrow XYZ \cdot \end{aligned}$$

- A produção vazia $A \rightarrow \epsilon$ tem apenas um item: $A \rightarrow \cdot$
- Intuitivamente, um item mostra quando da produção é que já foi reconhecida e o que é esperado a seguir.
- Assim o item $A \rightarrow \cdot XYZ$ mostra que esperamos vir a reconhecer uma derivação de XYZ .
- Já o item $A \rightarrow X \cdot YZ$ mostra que acabamos de “consumir” um sequência derivada de X e esperamos uma derivação de YZ .
- O item $A \rightarrow XYZ \cdot$ indica que o corpo da produção A está completo pelo que podemos proceder à sua redução.

Analizador ascendente LR(0) canónica

- Uma colecção de conjuntos de $LR(0)$ – designada $LR(0)$ canónica – fornece a base para se construir um autómato finito determinista que é utilizado nesta decisão.
- Este autómato é chamado autómato $LR(0)$.
- Cada estado deste autómato representa um conjunto de itens nessa colecção $LR(0)$ canónica.
- Para construir a colecção de conjuntos de itens, vamos definir uma gramática aumentada e duas novas funções: CLOSURE (fecho) e GOTO
- Se uma gramática G tiver S como símbolo inicial, então G' será a sua gramática aumentada, em que passará a existir um novo símbolo inicial (S') definido pela produção $S' \rightarrow S$
- Este novo símbolo serve para indicar a aceitação da entrada.

Analizador ascendente LR(0) canónica: função CLOSURE

- Se I é um conjunto de itens para uma gramática G , então $CLOSURE(I)$ é o conjunto de itens construído da seguinte forma:
 1. Todos os elementos de I pertencem a $CLOSURE(I)$.
 2. Se $A \rightarrow \alpha \cdot B\beta$ for um dos itens de $CLOSURE(I)$ e existir a produção $B \rightarrow \gamma$, então adicionar o item $B \rightarrow \cdot \gamma$ a $CLOSURE(I)$ caso ainda lá não exista. Aplicar esta regra até não existirem mais itens a ser adicionados a $CLOSURE(I)$.
- Considere a seguinte gramática aumentada (com E' como símbolo inicial):

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

- Se I é o conjunto de itens apenas com o elemento $\{[E' \rightarrow \cdot E]\}$, então:

$$\begin{aligned} CLOSURE(I) = \{ [E' \rightarrow \cdot E], [E \rightarrow \cdot E + T], [E \rightarrow \cdot T], [T \rightarrow \cdot T * F], \\ [T \rightarrow \cdot F], [F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}] \} \end{aligned}$$

Analisador ascendente LR(0) canónica: função GOTO

- Se I é um conjunto de itens para uma gramática G e X um símbolo de G , então $\text{GOTO}(I, X)$ é o fecho (*closure*) do conjunto de todos os itens $[A \rightarrow \alpha X \cdot \beta]$ desde que $[A \rightarrow \alpha \cdot X \beta]$ pertença a I .
- Se I é o conjunto de itens $\{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\}$, então:

$$\begin{aligned}\text{GOTO}(I, +) = & \{[E \rightarrow E \cdot + \cdot T], [T \rightarrow \cdot T * F], [T \rightarrow \cdot F], \\ & [F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}]\}\end{aligned}$$

Analisador ascendente LR(0) canónica: algoritmo

- O seguinte algoritmo permite a criação do autómato $LR(0)$ para a gramática aumentada G' :
 1. O estado inicial será $\text{CLOSURE}([S' \rightarrow \cdot S])$.
 2. Sempre que um novo estado é adicionado, verificar as transições que esse estado pode ter por ocorrência de símbolos (terminais ou não terminais).
 3. Sempre que dessas transições resultar um estado inexistente, criar esse novo estado, e repetir o processo.
- Note que este procedimento tem algumas semelhanças formais com a transformação de AFND em AFD (análise lexical, autómatos finitos).

Analisador ascendente LR(0) canónica: exemplo

- Recuperando a gramática aumentada de expressões aritméticas:

$$\begin{array}{lcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

- O estado inicial será:

$$\begin{aligned}I_0 = & \text{CLOSURE}(\{[E' \rightarrow \cdot E]\}) \\ = & \{[E' \rightarrow \cdot E], [E \rightarrow \cdot E + T], [E \rightarrow \cdot T], [T \rightarrow \cdot T * F], [T \rightarrow \cdot F], [F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}]\}\end{aligned}$$

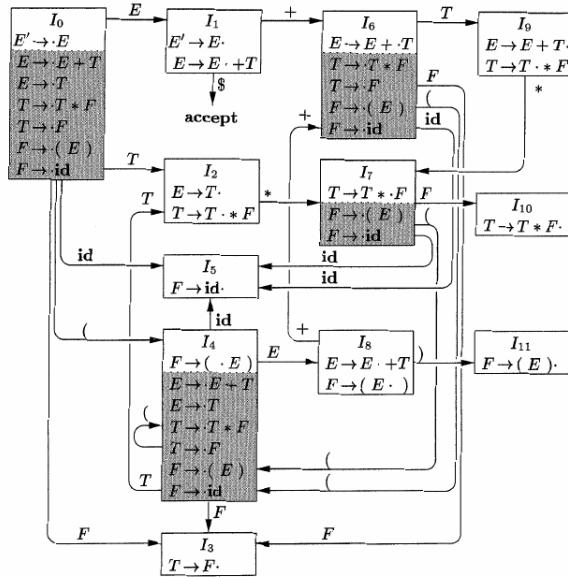
- Este estado pode evoluir pelos símbolos $E, T, F, (, \text{id}, \cdot$, logo:

$$\begin{aligned}I_1 = & \text{CLOSURE}(\text{GOTO}(I_0, E)) = \{[E' \rightarrow E \cdot], [E \rightarrow E \cdot + T]\} \\ I_2 = & \text{CLOSURE}(\text{GOTO}(I_0, T)) = \{[E \rightarrow T \cdot], [T \rightarrow T \cdot * F]\} \\ I_3 = & \text{CLOSURE}(\text{GOTO}(I_0, F)) = \{[T \rightarrow F \cdot]\} \\ I_4 = & \text{CLOSURE}(\text{GOTO}(I_0, ())) = \{[F \rightarrow (\cdot E)], [E \rightarrow \cdot E + T], [E \rightarrow \cdot T], \\ & [T \rightarrow \cdot T * F], [T \rightarrow \cdot F], [F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}]\} \\ I_5 = & \text{CLOSURE}(\text{GOTO}(I_0, \text{id})) = \{[F \rightarrow \text{id} \cdot]\}\end{aligned}$$

- As evoluções para os novos estados são:

$\text{GOTO}(I_1, \$) = \text{aceitar}$
 $I_6 = \text{CLOSURE}(\text{GOTO}(I_1, +)) = \{[E \rightarrow E + T], [T \rightarrow \cdot T * F], [T \rightarrow \cdot F], [F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}]\}$
 $I_7 = \text{CLOSURE}(\text{GOTO}(I_2, *)) = \{[T \rightarrow T * F], [F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}]\}$
 $I_8 = \text{CLOSURE}(\text{GOTO}(I_4, E)) = \{[E \rightarrow E + T], [F \rightarrow (E \cdot)]\}$
 $\text{CLOSURE}(\text{GOTO}(I_4, T)) = \{[E \rightarrow T \cdot], [T \rightarrow T \cdot * F]\} = I_2$
 $\text{CLOSURE}(\text{GOTO}(I_4, F)) = \{[T \rightarrow F \cdot]\} = I_3$
 $\text{CLOSURE}(\text{GOTO}(I_4, ())) = I_4$
 $\text{CLOSURE}(\text{GOTO}(I_4, \text{id})) = \{[F \rightarrow \text{id} \cdot]\} = I_5$
 $I_9 = \text{CLOSURE}(\text{GOTO}(I_6, T)) = \{[E \rightarrow E + T \cdot], [T \rightarrow T \cdot * F]\}$
 $\text{CLOSURE}(\text{GOTO}(I_6, F)) = \{[T \rightarrow F \cdot]\} = I_3$
 $\text{CLOSURE}(\text{GOTO}(I_6, ())) = I_4$
 $\text{CLOSURE}(\text{GOTO}(I_6, \text{id})) = \{[F \rightarrow \text{id} \cdot]\} = I_5$
 $I_{10} = \text{CLOSURE}(\text{GOTO}(I_7, F)) = \{[T \rightarrow T * F \cdot]\}$
 $\text{CLOSURE}(\text{GOTO}(I_7, ())) = I_4$
 $\text{CLOSURE}(\text{GOTO}(I_7, \text{id})) = \{[F \rightarrow \text{id} \cdot]\} = I_5$
 $I_{11} = \text{CLOSURE}(\text{GOTO}(I_8, ())) = \{[F \rightarrow (E \cdot)]\}$
 $\text{CLOSURE}(\text{GOTO}(I_9, *)) = I_7$

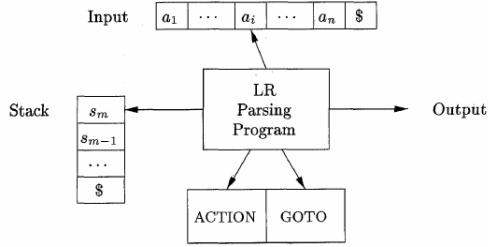
Analizador ascendente LR(0) canónica: exemplo - diagrama de transições
Donde resulta o seguinte diagrama de transições do autómato:



Analizador ascendente LR(0) canónica: exemplo - diagrama de transições (2)

- O conjunto de itens de cada estado pode ser dividido em dois:
 1. Itens nucleares: o item inicial e todos os itens em cujos o ponto não está à esquerda;
 2. Itens não-nucleares: todos os itens restantes (i.e. todos os itens em que o ponto está à esquerda, excepto o item inicial).
- Apenas é necessário guardar os itens nucleares (os outros, representados a sombreado na figura, estão na transição dos estados).

Analizador ascendente LR: algoritmo de reconhecimento



- A pilha contém os estados do autómato.
- A tabela de *parsing* tem duas partes: (1) uma função ACTION; (2) e uma função GOTO.
 - A função ACTION tem como argumentos um estado do autómato e um símbolo terminal (acrescida com o EOF).
 - O seu valor pode ter 4 formas: (1) deslocamento e transição para outro estado; (2) redução por uma produção; (3) aceitar a entrada; (4) rejeitar a entrada (erro).
 - A função GOTO indica as transições de estados no autómato em função de símbolos não terminais.

Analisador ascendente LR(0) canónica: exemplo - tabela de *parsing*

estado	acção					GOTO			
	id	+	*	()	\$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				aceitar			
2		$r_{E \rightarrow T}$	s_7		$r_{E \rightarrow T}$	$r_{E \rightarrow T}$			
3		$r_{T \rightarrow F}$	$r_{T \rightarrow F}$		$r_{T \rightarrow F}$	$r_{T \rightarrow F}$			
4	s_5			s_4			8	2	3
5		$r_{F \rightarrow id}$	$r_{F \rightarrow id}$		$r_{F \rightarrow id}$	$r_{F \rightarrow id}$			
6	s_5			s_4			9	3	
7	s_5			s_4				10	
8		s_6			s_{11}				
9		$r_{E \rightarrow E+T}$	s_7		$r_{E \rightarrow E+T}$	$r_{E \rightarrow E+T}$			
10		$r_{T \rightarrow T * F}$	$r_{T \rightarrow T * F}$		$r_{T \rightarrow T * F}$	$r_{T \rightarrow T * F}$			
11		$r_{F \rightarrow (E)}$	$r_{F \rightarrow (E)}$		$r_{F \rightarrow (E)}$	$r_{F \rightarrow (E)}$			

- As acções s_i indicam deslocamento para o estado i , e r_p redução pela produção p .
- Na aplicação do analisador, a pilha irá agora registar estados (em vez de símbolos).
- Sempre que exista uma transição com o *token* de entrada optar-se-á pelo seu deslocamento, colocando na pilha o estado para onde a transição é feita.
- Caso contrário, opta-se pela redução, fazendo o número de pops na pilha igual ao número de símbolos do corpo da produção reduzida, e colocando na pilha a cabeça dessa produção.
- Para ilustrar o funcionamento do analisador, vamos aplicá-lo à entrada **id * id**

	<i>pilha</i>	<i>símbolos</i>	<i>entrada</i>	<i>acções</i>
(1)	0	\$	id₁ * id₂ \$	deslocamento, <i>push 5</i>
(2)	0 5	\$ id₁	* id₂ \$	redução por $F \rightarrow \text{id}, 1 \times \text{pop}$
(3)	0	\$ <i>F</i>	* id₂ \$	<i>push(goto(0, F))</i>
(4)	0 3	\$ <i>F</i>	* id₂ \$	redução por $T \rightarrow F, 1 \times \text{pop}$
(5)	0	\$ <i>T</i>	* id₂ \$	<i>push(goto(0, T))</i>
(6)	0 2	\$ <i>T</i>	* id₂ \$	deslocamento, <i>push 7</i>
(7)	0 2 7	\$ <i>T</i> *	id₂ \$	deslocamento, <i>push 5</i>
(8)	0 2 7 5	\$ <i>T</i> * id₂	\$	redução por $F \rightarrow \text{id}, 1 \times \text{pop}$
(9)	0 2 7	\$ <i>T</i> * <i>F</i>	\$	<i>push(goto(7, F))</i>
(10)	0 2 7 10	\$ <i>T</i> * <i>F</i>	\$	redução por $T \rightarrow T * F, 3 \times \text{pop}$
(11)	0	\$ <i>T</i>	\$	<i>push(goto(0, T))</i>
(12)	0 2	\$ <i>T</i>	\$	redução por $E \rightarrow T, 1 \times \text{pop}$
(13)	0	\$ <i>E</i>	\$	<i>push(goto(0, E))</i>
(14)	0 1	\$ <i>E</i>	\$	aceitar

6.4.2 Outros analisadores ascendentes LR

- Os analisadores ascendentes gerados por ferramentas tipo *yacc/bison* não são analisadores $LR(0)$ simples.
- Ao contrário deste, esses analisadores fazem uso da antevista do próximo token de entrada e são designados por $LALR(1)$.
- A utilização dessa informação permite uma melhor escolha das transições a serem feitas, e alargam enormemente o leque de gramáticas independentes de contexto que podem ser implementadas.
- A sua implementação directa é muito trabalhosa, pelo que não iremos detalhar a sua implementação.

