UDESC

Compiladores

Cristiano Damiani Vasconcellos
cristiano.vasconcellos@udesc.br

UDESC

Bibliografia Recomendada

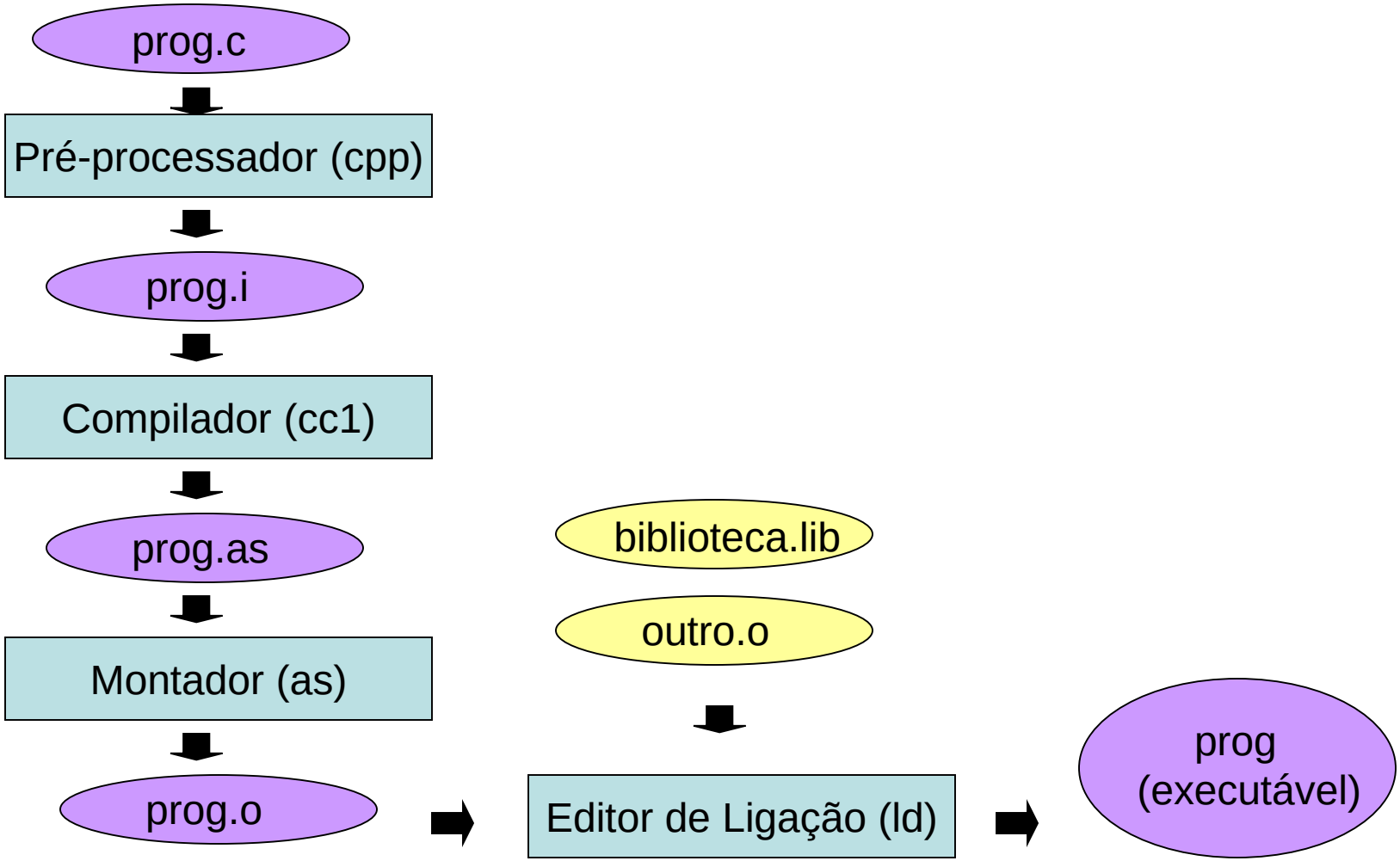
Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D.; Compiladores: Princípios, Técnicas e Ferramentas. Pearson.

Cooper, Keith D.; Torczon, Linda.; Construindo compiladores. Elsevier.

Bryant, Randal E.; O'Hallaron, David R.; Computer Systems: A Programmer's Perspective. Prentice Hall.

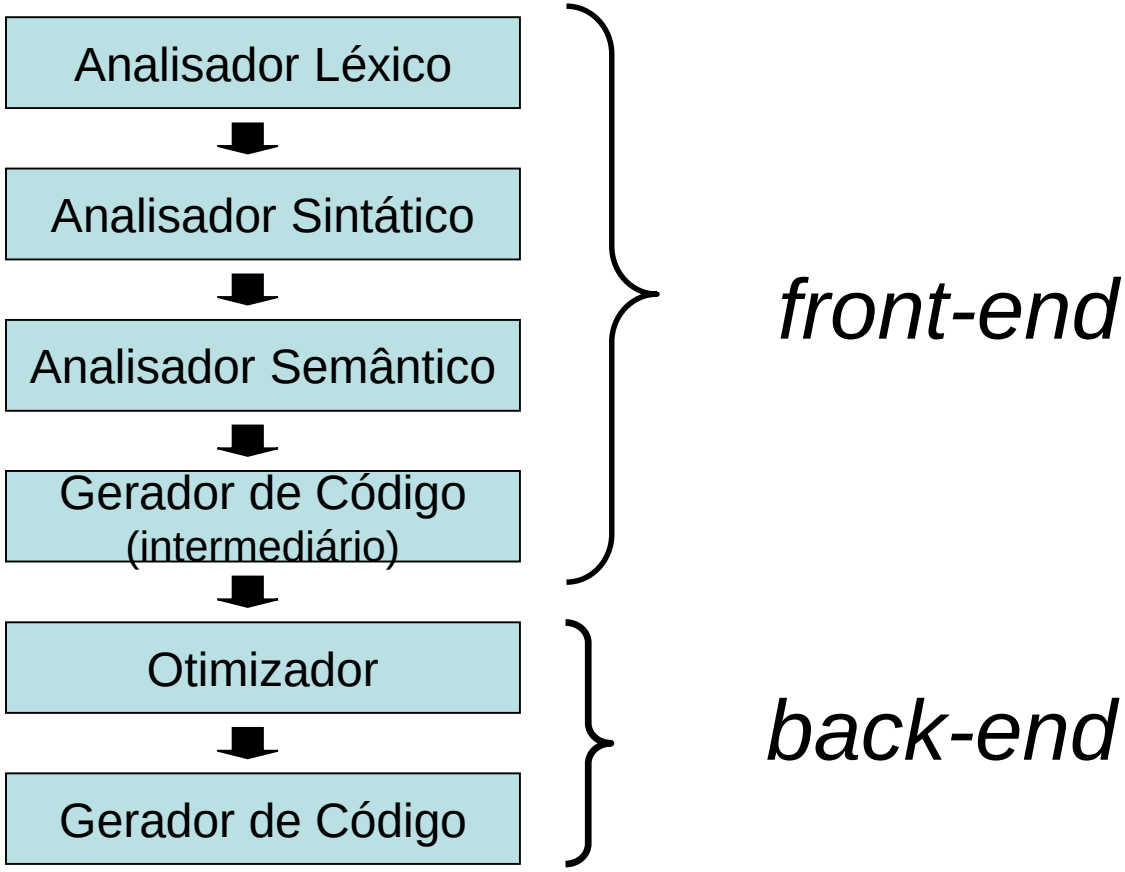
UDESC

Introdução
(Construção de um Executável)



UDESC

Introdução
(Fases de um Compilador)



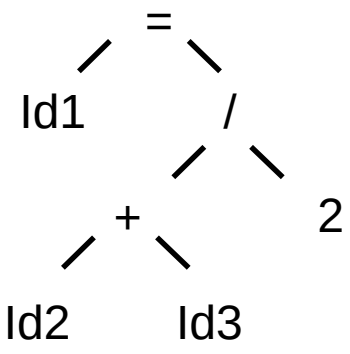
Introdução

final = (nota1 + nota2) / 2;

Analisador Léxico

Id1 = (Id2 + Id3) / 2

Analisador Sintático



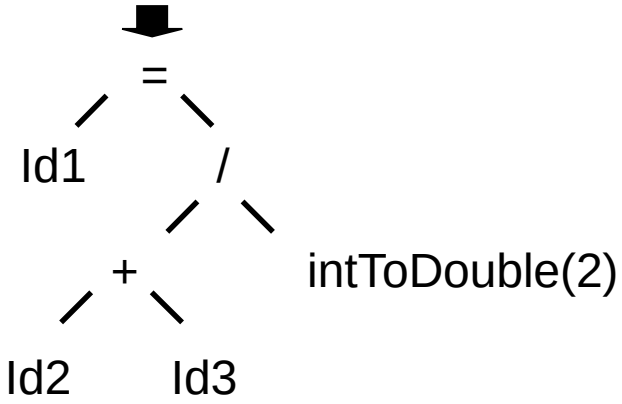
Árvore Sintática

Tabela de Símbolos

Id1	final	double	...
Id2	nota1	double	...
Id3	nota2	double	...
...			

Introdução

Analisador Semântico



Gerador de Código (intermediário)

temp1 = Id2 + Id3

temp2 = temp1 / 2.0

Id1 = temp2

Tabela de Símbolos

Id1	final	double	**...**
Id2	nota1	double	**...**
Id3	nota2	double	**...**
...			

UDESC

Análise Léxica

O Analisador Léxico (*scanner*) examina o programa fonte caractere por caractere agrupando-os em conjuntos com um significado coletivo (*tokens*):

- palavras chave (*if, else, while, int, etc*),
- operadores (+, -, *, /, ^, &&, *etc*),
- constantes (*1, 1.0, ‘a’, 1.0f, etc*),
- literais (*“Alo Mundo”, etc*),
- símbolos de pontuação (*; , {, }, etc*),
- labels.

UDESC

Análise Léxica

constanteInt $\rightarrow$ digito digito*
constanteDouble $\rightarrow$ digito digito*. digito*

digito $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

X Representa uma seqüência de zero ou mais X.*

UDESC

Análise Sintática

Verifica se as frases obedecem as regras sintáticas da linguagem:

Por exemplo, uma expressão pode ser definida como:

expressão + expressão

expressão – expressão

(expressão)

constante

UDESC

Gramáticas Livres de Contexto

Definidas por uma quádrupla (V_N, V_T, S, P), onde:

V_N é um conjunto de símbolos não terminais (representam as construções sintáticas da linguagem).

V_T é um conjunto de símbolos terminais (*tokens* da linguagem).

$S \in V_N$ é o símbolo inicial da gramática.

P é um conjunto de regras de produção, pares ordenados representados na forma $\alpha \rightarrow \beta$, onde $\alpha \in V_N$ e $\beta \in (V_N \cup V_T)^*$.

UDESC

Gramáticas Livres de Contexto

<expr> → <expr> + <expr>
| <expr> – <expr>
| (<expr>)
| <const>

<const> → <const><const>
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9

UDESC

Derivação

Verificar se uma frase faz parte da linguagem gerada pela gramática, envolve sucessivas substituições dos símbolos que ocorrem do lado esquerdo da produção pela sua construção sintática correspondente.

Essa substituição é chamada derivação sendo normalmente denotada pelo símbolo $\Rightarrow$. E deve iniciar a partir do símbolo inicial da gramática.

UDESC

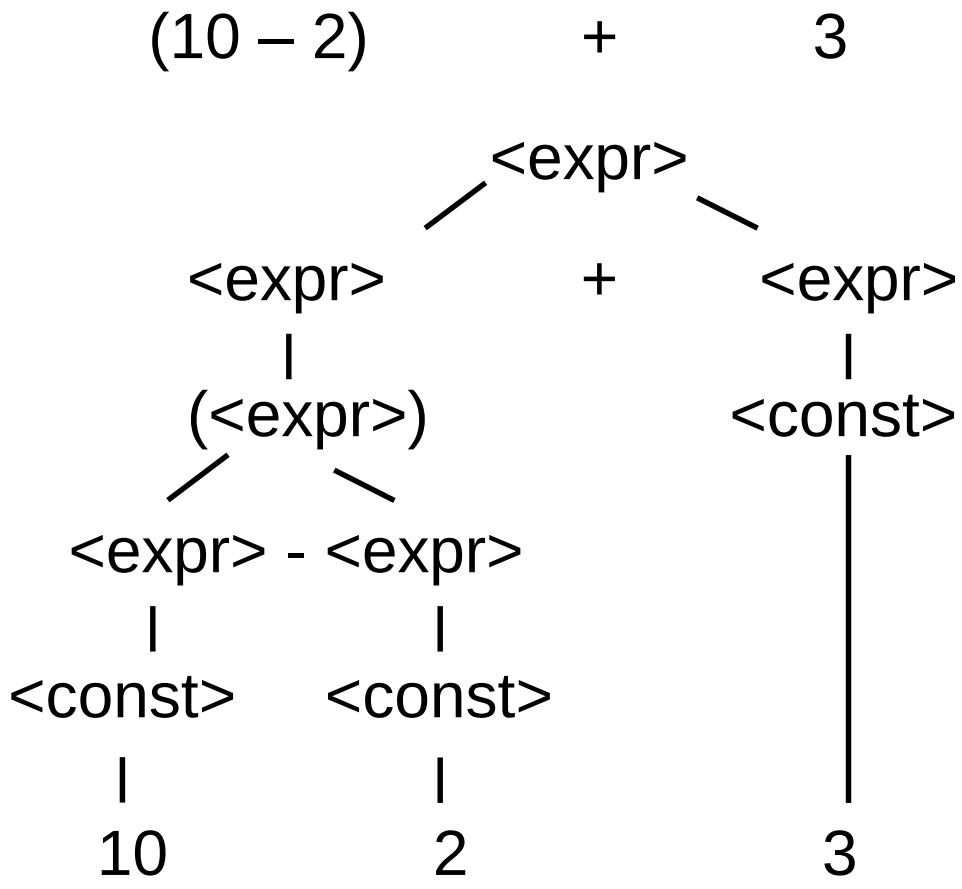
Derivação

<expressão>

⇒ <expr> + <expr>
⇒ (<expr>) + <expr>
⇒ (<expr> - <expr>) + <expr>
⇒ (<const> - <expr>) + <expr>
⇒ (<const><const> - <expr>) + <expr>
⇒ (1<const> - <expr>) + <expr>
⇒ (10 - <expr>) + <expr>
⇒ (10 - <const>) + <expr>

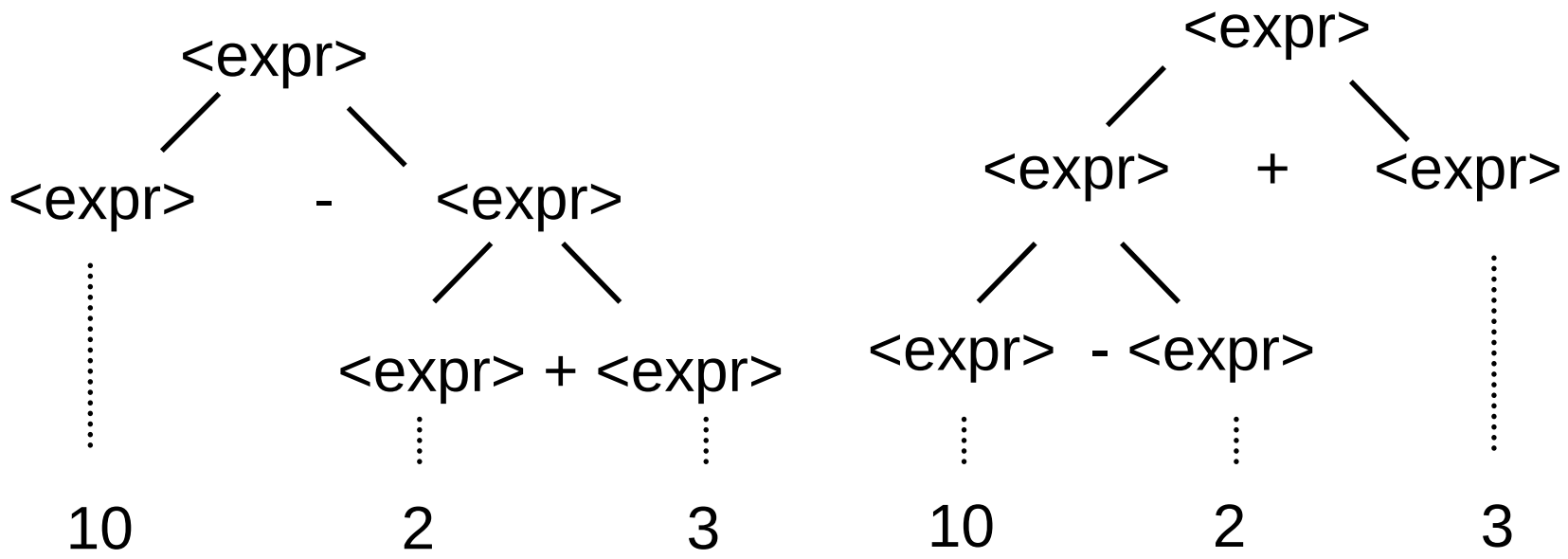
...
⇒ (10 - 2) + 3

Árvore Sintática de Derivação (*Parser Tree*)



Gramáticas Ambíguas

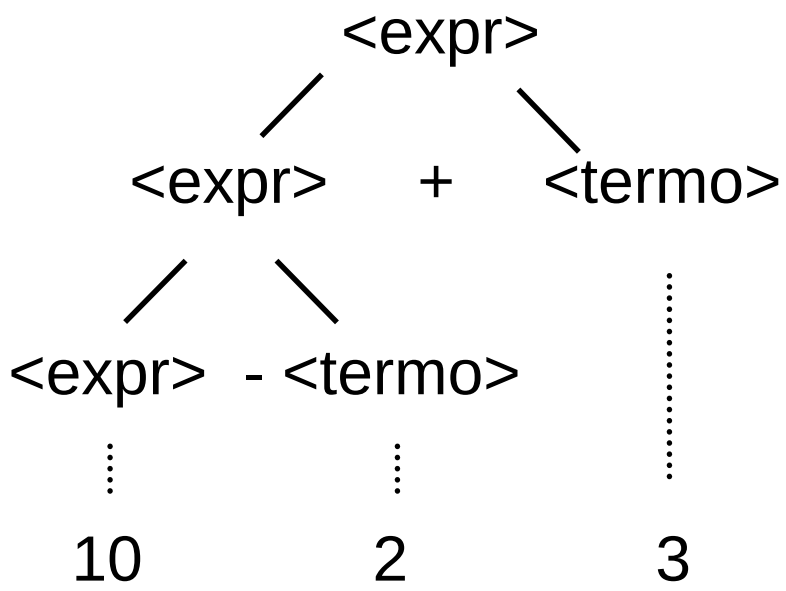
10 – 2 + 3



Gramáticas

<expr> → <expr> + <termo>
| <expr> - <termo>
| <termo>

<termo> → (<expr>)
| <const>



<expr>

⇒ <expr> + <termo>

⇒ <expr> - <termo> + <termo>

⇒ <termo> - <termo> + <termo>

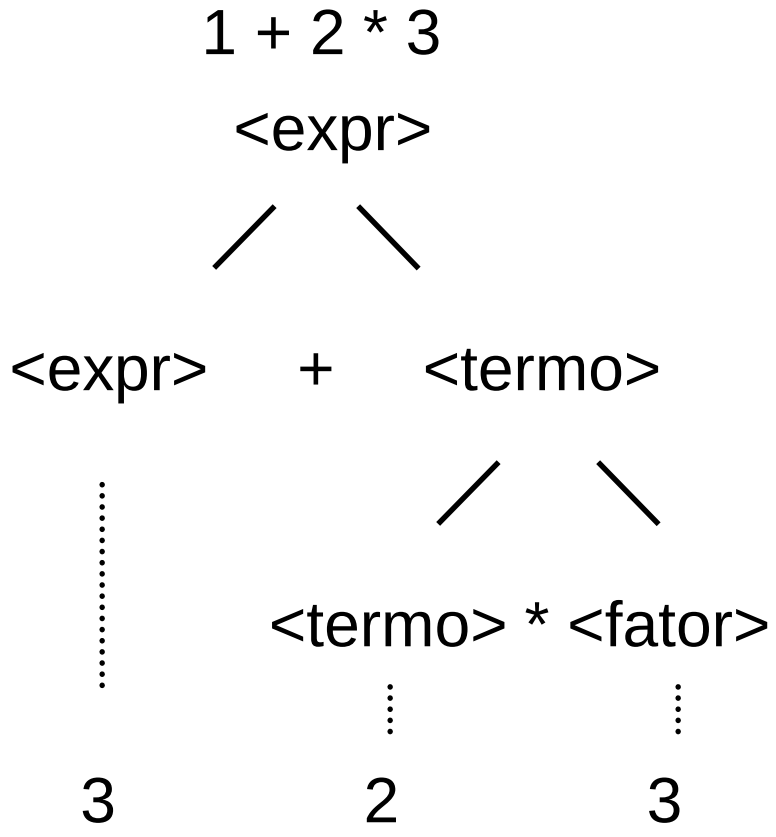
⇒ 10 – 2 + 3

Gramáticas

<expr> → <expr> + <termo>
| <expr> - <termo>
| <termo>

<termo> → <termo> * <fator>
| <termo> / <fator>
| <fator>

<fator> → (<expr>)
| <const>



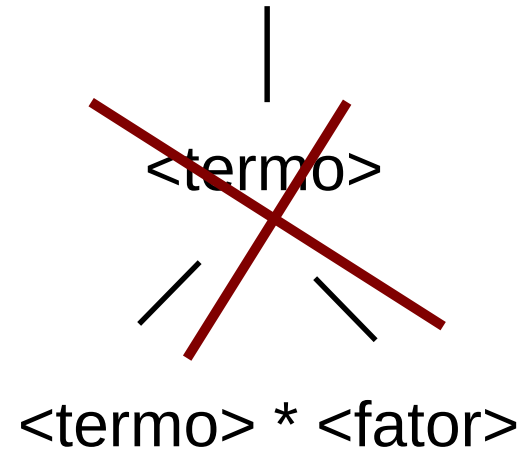
Gramáticas

<expr> → <expr> + <termo>
| <expr> - <termo>
| <termo>

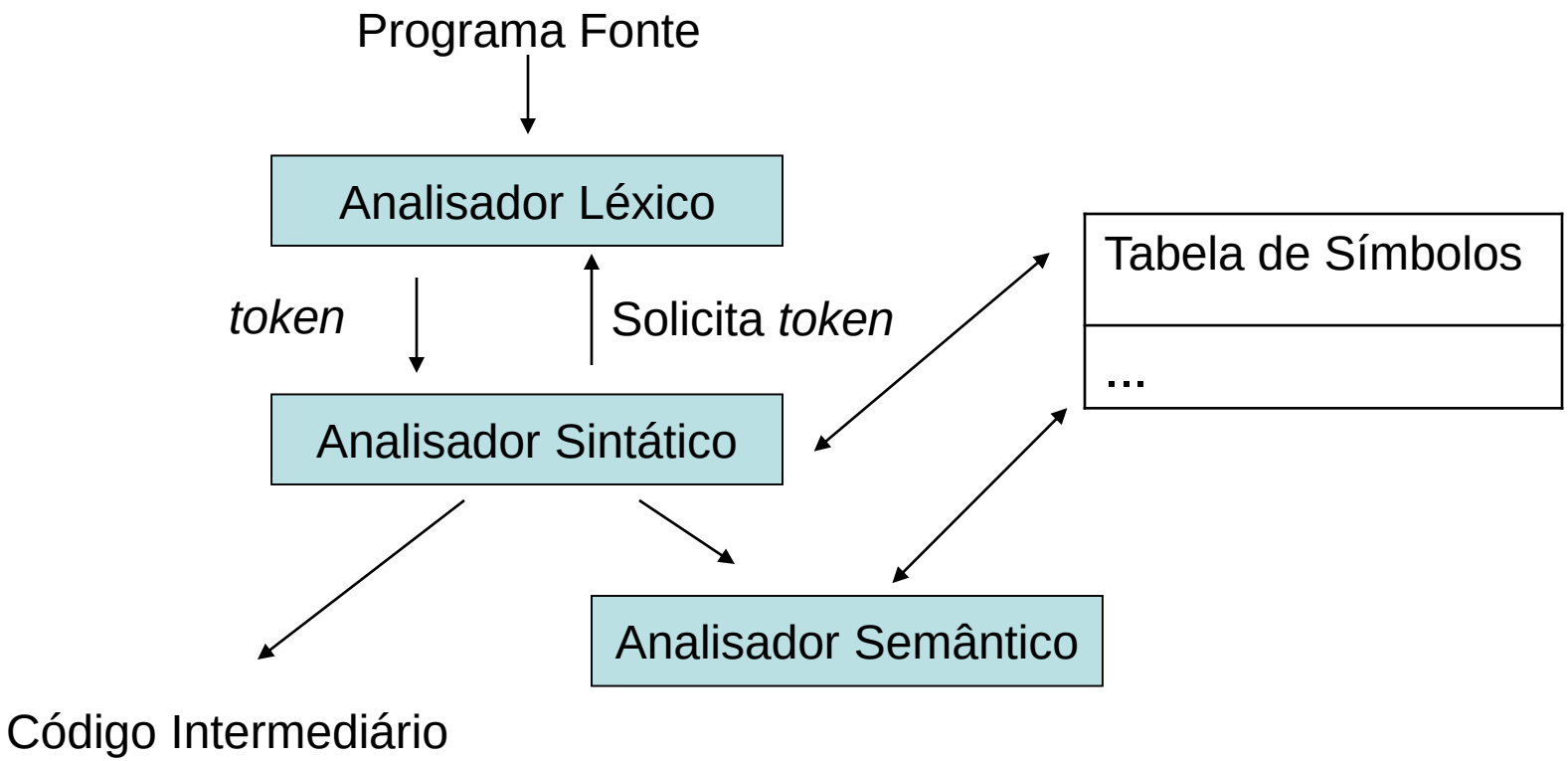
<termo> → <termo> * <fator>
| <termo> / <fator>
| <fator>

<fator> → (<expr>)
| <const>

1 + 2 * 3



Tradução Dirigida pela Sintaxe



UDESC

Gramáticas - Exercícios

1. Considerando a gramática apresentada anteriormente derive as expressões e apresente a árvore sintática correspondente:
   (1 + 2) * 3
   (1 – 2) + 3 * 4
2. Altere a gramática para incluir o operador unário -, esse operador deve ter precedência maior que todos os outros operadores.
3. Altere a gramática para que os operadores de adição, subtração, multiplicação e divisão tenham associatividade da direita para a esquerda.
4. Defina uma gramática para expressões aritméticas (operadores +, -, *, /) pós fixadas.

UDESC

Gramáticas

Dados 2 conjuntos independentes de símbolos:

- V_T – Símbolos terminais.
- V_N – Símbolos não terminais.

Uma gramática é definida como a quádrupla:

$$(V_N, V_T, S, P)$$

$S \in V_N$ é o símbolo inicial da gramática.

P é um conjunto de regras de reescrita na forma:

$\alpha \rightarrow \beta$, sendo: $\alpha \in (V_N \cup V_T)^* V_N (V_N \cup V_T)^*$

$$\beta \in (V_N \cup V_T)^*$$

Classificação de Gramáticas

- Irrestritas – nenhuma restrição é imposta
- Sensíveis ao Contexto - $|\alpha| \leq |\beta|$
- Livres de Contexto - $\alpha \in V_N$
  $\beta \in (V_N \cup V_T)^+$
- Regulares - $\alpha \in V_N$
  β tem *a* forma a ou *aB*, sendo
  $a \in V_T$ e $B \in V_N$

UDESC

Gramáticas Regulares

Uma gramática regular gera uma linguagem regular.

$C \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 9$
$\quad \mid 0C \mid 1C \mid 2C \mid 3C \mid 4C \mid 5C \mid 7C \mid 8C \mid 9C$

$C \rightarrow CC \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 9$

Linguagens Regulares

- Geradas a partir de uma gramática regular;
- Podem ser representadas por meio de uma expressão regular;
- Podem ser reconhecidas por Autômatos Finitos.

Considerando linguagens compostas por símbolos 0 e 1 podemos afirmar:

a linguagem $L_1 = \{0^n 1^n \mid n \geq 1\}$ não é regular;
a linguagem $L_2 = \{0^n 1^m \mid n \geq 1, m \geq 1\}$ é regular;

UDESC

Expressões Regulares

Maneira compacta de representar linguagens regulares. É composta de 3 operações. Sendo e_1 e e_2 expressões que geram respectivamente duas linguagens regulares L_1 e L_2:

- Concatenação: $e_1 e_2 = \{ xy \mid x \in L_1 \text{ e } y \in L_2 \}$
- Alternação: $e_1 | e_2 = \{ x \mid x \in L_1 \text{ ou } x \in L_2 \}$
- Fechamento: e_1^* = *zero ou mais ocorrências de* e_1.

É definida a precedência desses operadores como sendo: fechamento, concatenação, alternação (da maior precedência para a menor).

UDESC

Expressões Regulares

Exemplos:

identificador $\rightarrow$ (letra | _) (letra | digito | _)*

letra $\rightarrow$ a | b | ... | z | A | B | ... | Z

digito $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

constInt $\rightarrow$ digito digito*

constDouble $\rightarrow$ digito digito*.digito* | . digito digito*

UDESC

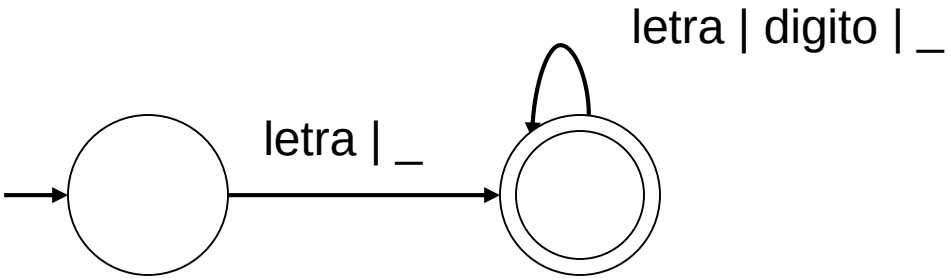
Autômato Finito

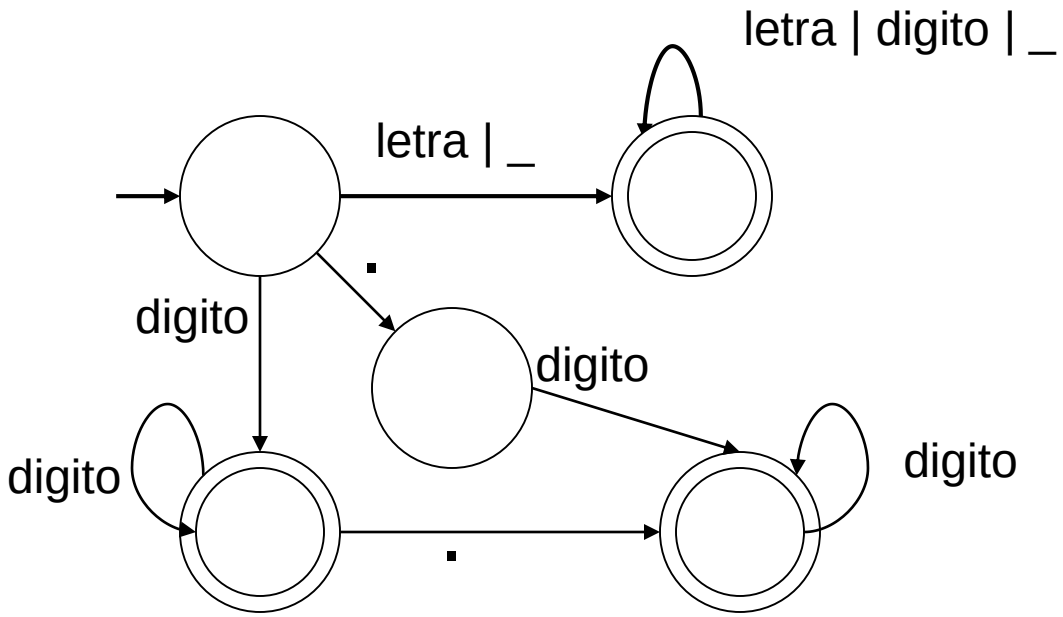
A linguagem gerada por uma gramática regular pode ser reconhecida por um autômato finito.

Um autômato finito consiste em:

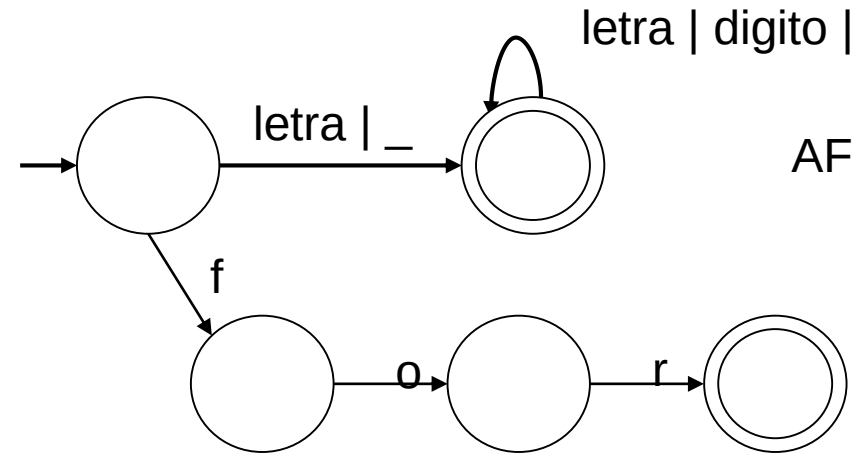
1. Um conjunto finito de estados.
2. Um conjunto finito de símbolos de entrada (alfabeto).
3. Uma função de transição que tem como argumentos um estado e um símbolo de entrada e retorna a um estado.
4. Um estado inicial.
5. Um conjunto de estados finais também chamados estados de aceitação.

Autômato Finito

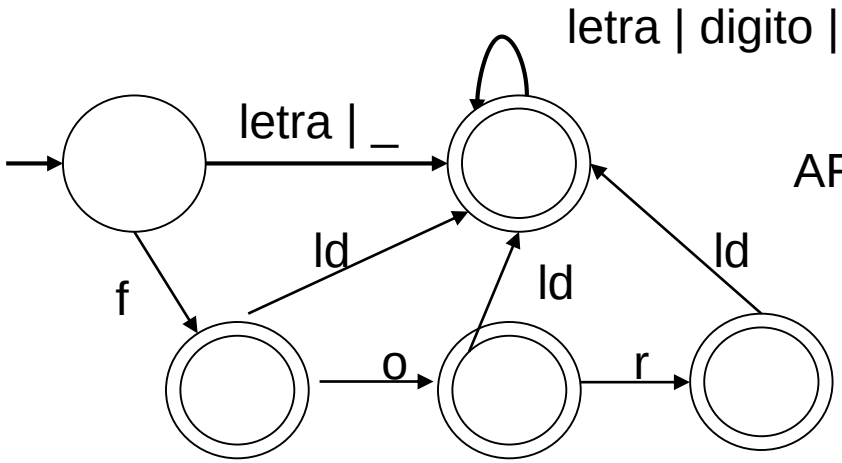




Autômato Finito



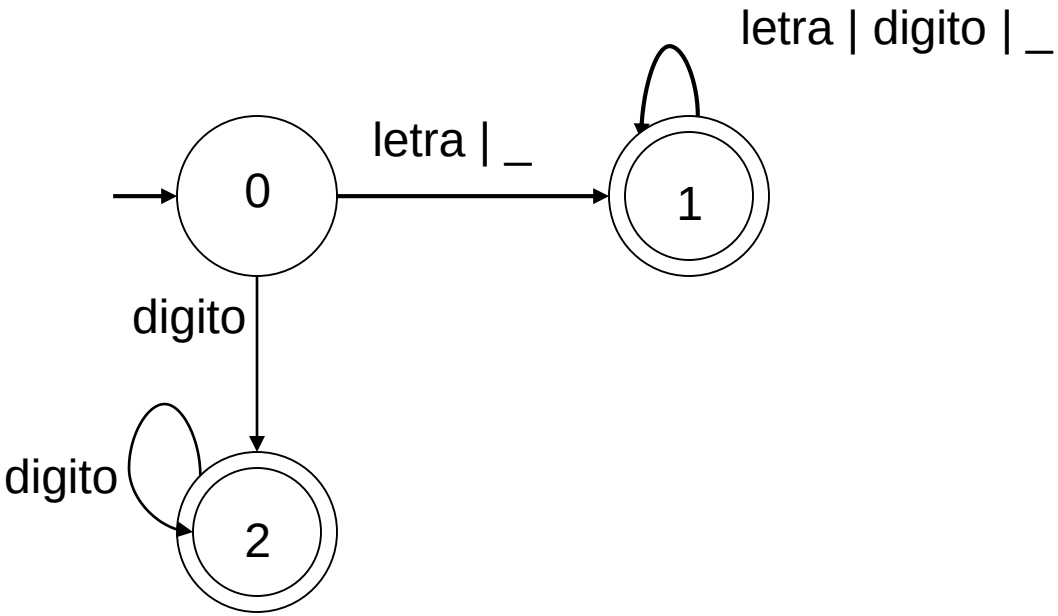
AFN – Autômato Finito Não Determinista



AFD – Autômato Finito Determinista

Onde ld representa letra | digito | _ (com exceção das letras que fazem transições para outros estados).

Autômato Finito Implementação



Geradores de Analisadores Léxicos

```
delim   [ \t]
ws      {delim}+
letra   [A-Za-z]
digito  [0-9]
id      {letra}({letra}|{digito})*
int     {digito}+
real    {digito}+\.{digito}*(E[+-]?{digito}+)?
char    '{letra}'
string  '({letra}|{digito}|[ \t\\:])*'
%%
{char}  {yylval.ptr=insereTab(&TabSimb[0], yytext);return TCCHARACTER;}
{string} {yylval.ptr=insereTab(&TabSimb[0], yytext);return TCSTRING;}
\n       {num_linhas++;}
FUNCTION {return TFUNCTION;}
INTEGER  {return TINTEGER;}
ARRAY    {return TARRAY;}
IF       {return TIF;}
{id}     {yylval.ptr=instalar(yytext); return TID;}
"<"      {return TMENOR;}
```

Análise Léxica - Exercícios

1. Escreva uma gramática, expressão regular e AFD que defina os números binários terminados em zero.
2. Mostre uma expressão regular e um AFD correspondentes a gramática abaixo:
   $S \rightarrow aS \mid aB \quad B \rightarrow bC \quad C \rightarrow aC \mid a$
3. Escreva uma expressão regular para as constantes *double* da linguagem C.

UDESC

Analisador Sintático

Obtém uma sequência de *tokens* fornecida pelo analisador léxico e verifica se a mesma pode ser gerada pela gramática.

Os métodos de análise sintática comumente usados em compiladores são classificados como:

- Métodos *top-down* (descendente).
- Métodos *bottom-up* (ascendente).

Os métodos eficientes, tanto *top-down* quanto *bottom-up*, trabalham com subconjuntos das gramáticas livres de contexto.

UDESC

Métodos *top-down*

Podem ser vistos como a tentativa de encontrar a derivação mais a esquerda para uma cadeia de entrada. Partindo do símbolo inicial da gramática são aplicadas sucessivas derivações tentado produzir a cadeia que se deseja reconhecer.

Exemplos:

- Método descendente recursivo.
- Método LL(1).

UDESC

Método Descendente Recursivo

<expr> → + <expr> <expr>

| - <expr> <expr>

<expr> → <const>

<const> → 0 | 1| 2 | 3| 4 | 5 | 6 | 7 | 8| 9

Método Descendente Recursivo

```
void cons()
{
        if (isdigit(lookahead))
            nextToken();
        else
            erro("Erro sintático");
}
void expr ()
{
        if (lookahead == '+' || lookahead == '-')
        {
            nextToken();  expr(); expr();
        }
        else
            cons();
}
```

UDESC

Analisadores Sintáticos Preditivos

Escrevendo a gramática de forma cuidadosa, podemos obter uma gramática processável por um analisador sintático que não necessite de retrocesso. Dado um símbolo de entrada *a* e um não terminal *A,* a ser expandido, a gramática deve possuir uma única produção que leve ao reconhecimento da cadeia iniciada com *a*.

Analisadores sintáticos não preditivos (ou não deterministas) necessitam de retrocesso (*backtraking*) e, em geral, são ineficientes.

Fatoração à Esquerda

As vezes é necessário fazer alterações na gramática que possibilitem a implementação de um reconhecedor preditivo:

<cmd> → if <expr> then <cmd> else <cmd>

| if <expr> then <cmd>

<cmd> → if <expr> then <cmd><cmd'>

<cmd'> → else <cmd>

| ε

UDESC

Fatoração à Esquerda

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Eliminação da Recursividade à Esquerda

$E \rightarrow E + T$

$\quad | \; E - T$

$\quad | \; T$

$T \rightarrow c$

$\quad | \; (E)$

E

$\Rightarrow E - T$

$\Rightarrow E + T - T$

$\Rightarrow T + T - T$

$^{*}\Rightarrow c + c - c$

Eliminação da Recursividade à Esquerda

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

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

$$T \rightarrow c \mid (E)$$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid -TE' \mid \varepsilon$$

$$T \rightarrow c \mid (E)$$

Análise Sintática Preditiva não Recursiva LL(1)

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$\quad | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow * FT'$

$\quad | \varepsilon$

$F \rightarrow c$

$\quad | (E)$

Não Terminal	c	+	*	(	)	#
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	* FT’		ε	ε
F	c			(E)		

E

$\Rightarrow$ TE’

$\Rightarrow$ FT’E’

$\Rightarrow$ cT’E’

$\Rightarrow$ cE’

$\Rightarrow$ c+TE’

$\Rightarrow$ c+FT’E’

$\Rightarrow$ c+cT’E’

$\Rightarrow$ c+c*FT’E’

$\Rightarrow$ c+c*cT’E’

$\Rightarrow$ c+c*cE’

$\Rightarrow$ c+c*c

Analisador Sintático LL(1)

Considerando *w* a cadeia de entrada.
Empilhar #, Empilhar o símbolo inicial da gramática.
Faça *p* apontar para o primeiro símbolo de w#
Repetir
    Seja *X* o símbolo no topo da pilha e *a* o símbolo apontado por *p*;
    Se *X* for um terminal ou # então
        Se *X* = *a* então
            Remover *X* da pilha e avançar *p*;
        Senão erro.
    Senão /* *X* não é um terminal */
        Se M[*X*, *a*] = $X \rightarrow Y_1 Y_2 ... Y_k$ então
            Remover *X* da Pilha
            Empilhar $Y_k ... Y_2 Y_1$
        Senão
            erro
Até que *X* = #

UDESC

Analisador Sintático LL(1)

Uma gramática cuja tabela não possui entradas multiplamente definidas é dita LL(1). O primeiro L indica a varredura da cadeia de entrada, que e feita da esquerda para a direita (*left to right*) o segundo L indica que são aplicadas derivações mais a esquerda (*left linear*). O número 1 indica que é necessário apenas um símbolo para decidir qual produção aplicar (*1 lookahead*).

Construção da Tabela LL(1)

A construção de um analisador sintático preditivo e auxiliada por duas funções associadas a gramática: PRIMEIROS e SEGUINTES (FIRST e FOLLOW).

Seja α uma cadeia qualquer de símbolos gramaticais, PRIMEIROS(α) representa o conjunto de símbolos terminais que começam as cadeias derivadas a partir de α. Se α $*\Rightarrow \varepsilon$ então ε também é um elemento de PRIMEIROS(α).

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow c$

PRIMEIROS(E) = { (, c}

$E \Rightarrow T$
$\Rightarrow F$
$\Rightarrow (E)$

$E \Rightarrow T$
$\Rightarrow F$
$\Rightarrow c$

Construção da Tabela LL(1)

SEGUINTES(A), para um não terminal A, é o conjunto de terminais *a* tais que existe uma derivação $S \overset{*}{\Rightarrow} \alpha A a \beta$, para alguma cadeia α e alguma cadeia β, onde S é o símbolo inicial da gramática. Ou seja o conjunto de símbolos que podem ocorrer após o não terminal A em alguma forma sentencial da gramática.

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow c$

SEGUINTES(F) = { +, #, *,) }

$E \Rightarrow E + T$
$\Rightarrow T + T$
$\Rightarrow F + T$

$E \Rightarrow E + T$
$\Rightarrow E + T$
$\Rightarrow E + F\ \#$

$E \Rightarrow T$
$\Rightarrow T * F$
$\Rightarrow F * F$

$E \Rightarrow T$
$\Rightarrow F$
$\Rightarrow (E)$
$\Rightarrow (E + T)$
$\Rightarrow (E + F)$

Construção da Tabela LL(1)

Entrada: Gramática

Saída: Tabela *M*

Para cada produção $A \rightarrow \alpha$ da gramática faça:

- Para cada terminal *a* em PRIMEIROS(α), adicione $A \rightarrow \alpha$ em M[*A*, *a*].
- Se ε estiver em PRIMEIROS(α), adicione $A \rightarrow \alpha$ em M[*A*, *b*], para cada terminal *b* em SEGUINTES(*A*).

Cada entrada indefinida em *M* indica uma situação de erro.

Construção da Tabela LL(1)

E → TE’

E’ → +TE’

| ε

T → FT’

T’ → * FT’

| ε

F → c

| (E)

PRIMEIROS (TE’) = {c, (}

PRIMEIROS (+TE’) = {+ }

SEGUINTES (E’) = {), # }

PRIMEIROS (FT’) = {c, (}

PRIMEIROS (*FT’) = { * }

SEGUINTES (T’) = { +,), # }

PRIMEIROS (c) = {c}

PRIMEIROS(E) = { (}

Não Terminal	c	+	*	(	)	#
E	**TE’**			**TE’**		
E’		**+TE’**			**ε**	**ε**
T	**FT’**			**FT’**		
T’		**ε**	*** FT’**		**ε**	**ε**
F	**c**			**(E)**		

UDESC

Métodos *bottom-up*

Podem ser vistos como a tentativa de se reduzir a cadeia de entrada ao símbolo inicial da gramática.

Exemplos:

- Precedência de Operadores;
- SLR(1), LR(1), LALR(1).

UDESC

Métodos LR(*k*)

Os métodos de análise sintática LR executam uma derivação mais a direita ao contrário. O L significa que a varredura da entrada e feita da esquerda para a direita (*left to right*), o R que a derivação correspondente é a derivação mais a direita (*rightmost derivation*) e o *k* indica o número de símbolos de entrada que tem que ser examinados para se tomar uma decisão na análise sintática.

A diferença entre os métodos SLR e LALR é apenas a técnica usada para a construção das tabelas sintáticas.

Métodos LR

Estado	AÇÃO						DESVIO		
	c	+	*	(	)	#	E	T	F
0	5			4			1	2	3
1		6				AC			
2		R2	7		R2	R2			
3		R4	R4		R4	R4			
4	5			4			8	2	3
5		R6	R6		R6	R6			
6	5			4				9	3
7	5			4					10
8		6			11				
9		R1	7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow c$

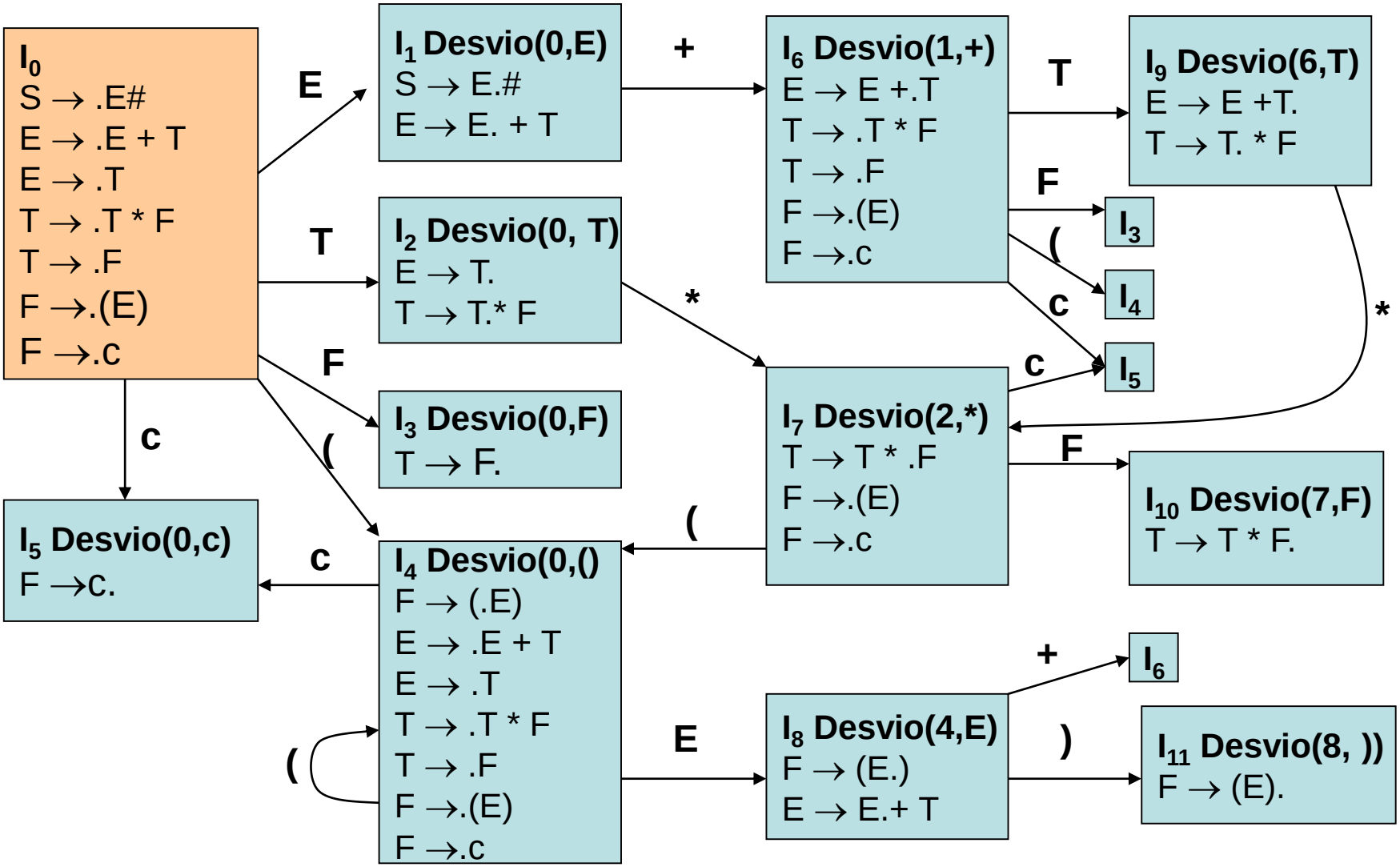
Algoritmo LR(1)

Considerando *w* a cadeia de entrada.
Empilhar 0. /* Estado inicial */
Faça *p* apontar para o primeiro símbolo de *w#*
Repetir para sempre
    Seja *s* o estado no topo da Pilha e *a* o símbolo apontado por p
    Se AÇÃO[*s*, *a*] = empilhar *s'* então
        Empilhar *a*; Empilhar *s'*;
        Avançar *p*;
    Senão
        Se AÇÂO[*s*, *a*] = reduzir $A \rightarrow \beta$ então
            Desempilhar 2 * $|\beta|$ símbolos;
            Seja *s'* o estado no topo da pilha
            Empilhar A; Empilhar DESVIO[*s'*, *A*];
        Senão
            Se AÇÃO[*s*, *a*] = aceitar então Retornar;
            Senão erro
fim

UDESC

Pilha : 0

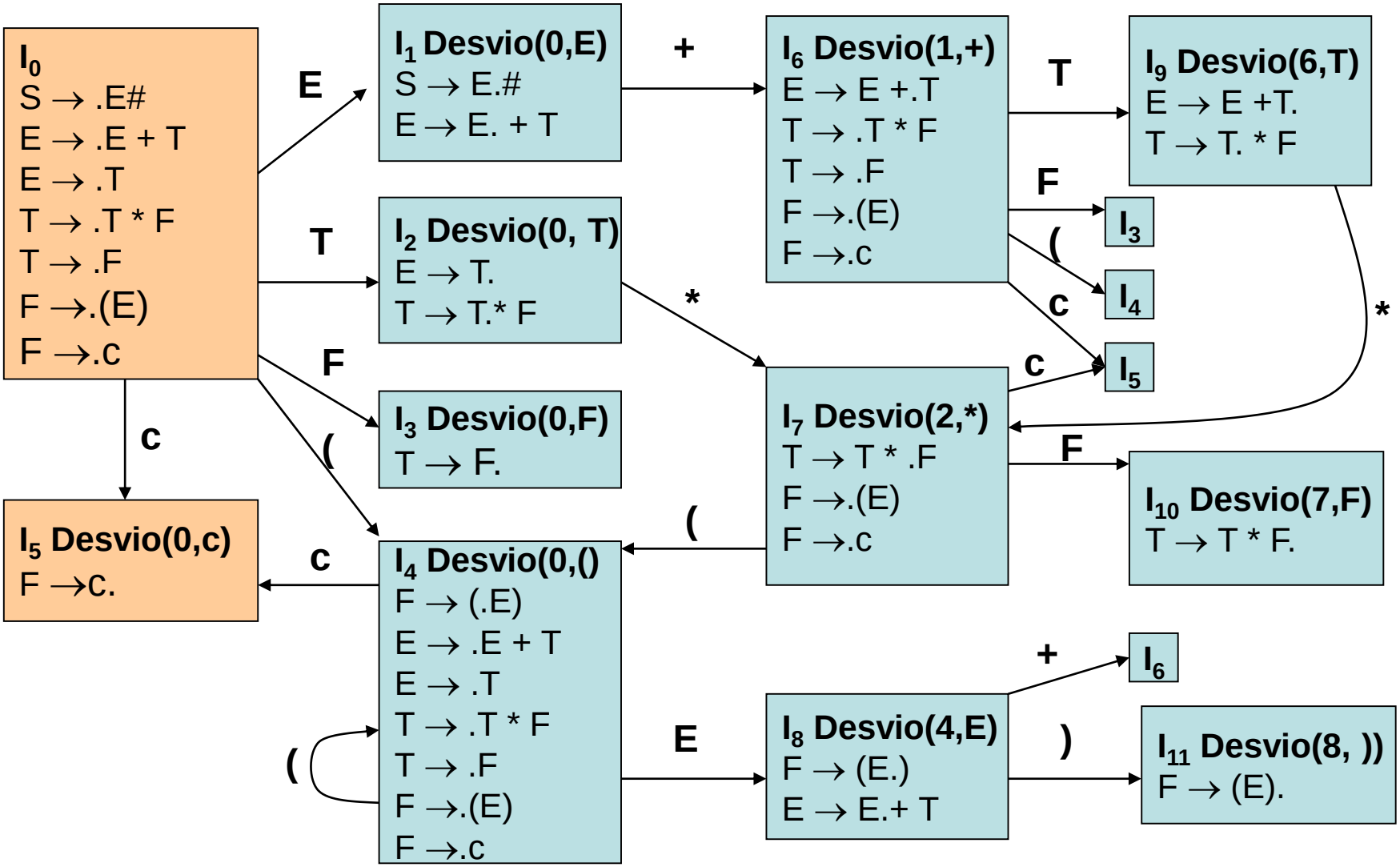
Entrada: c + c #



UDESC

Pilha : 0 c 5

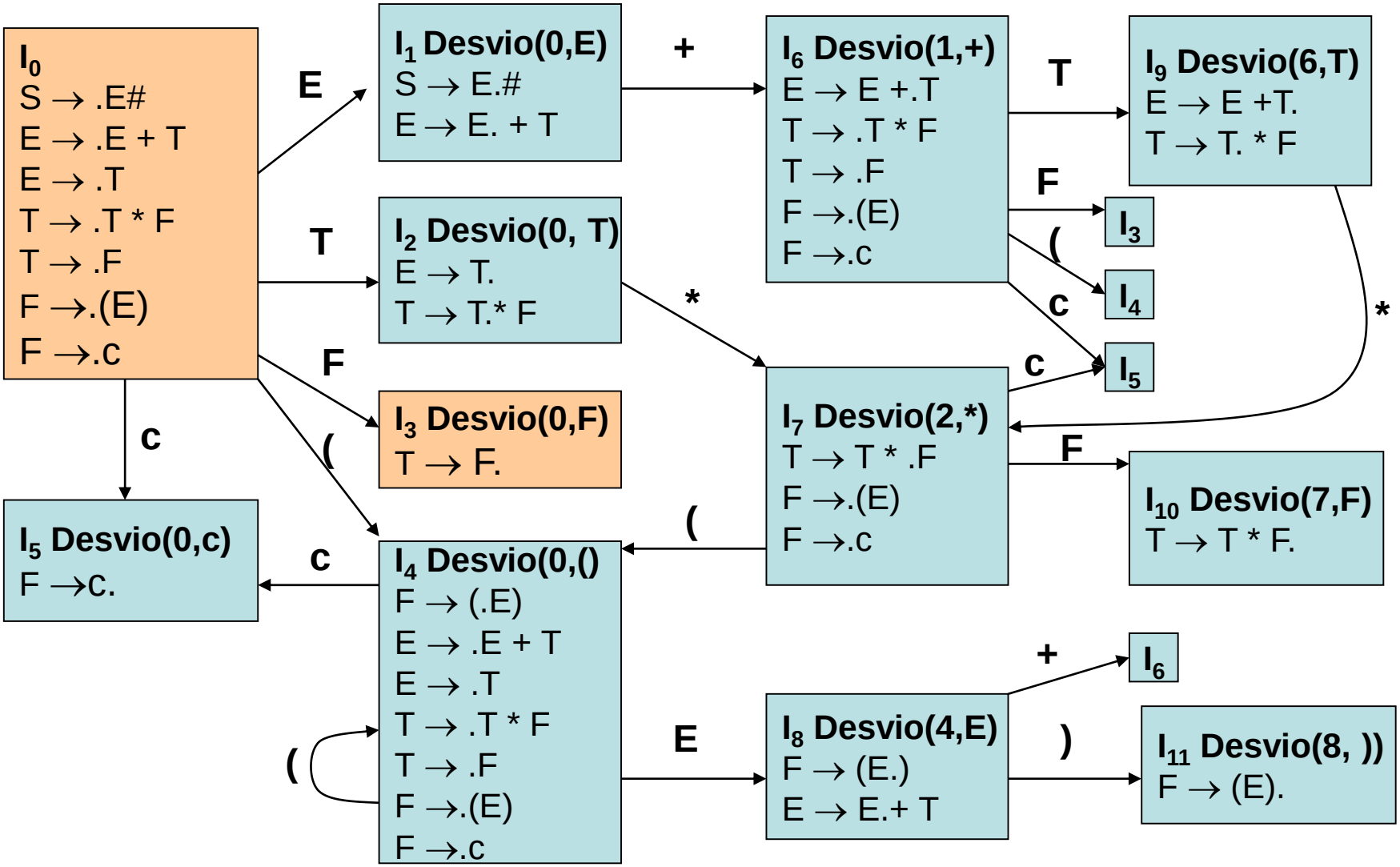
Entrada: c + c #



UDESC

Pilha : 0 F 3

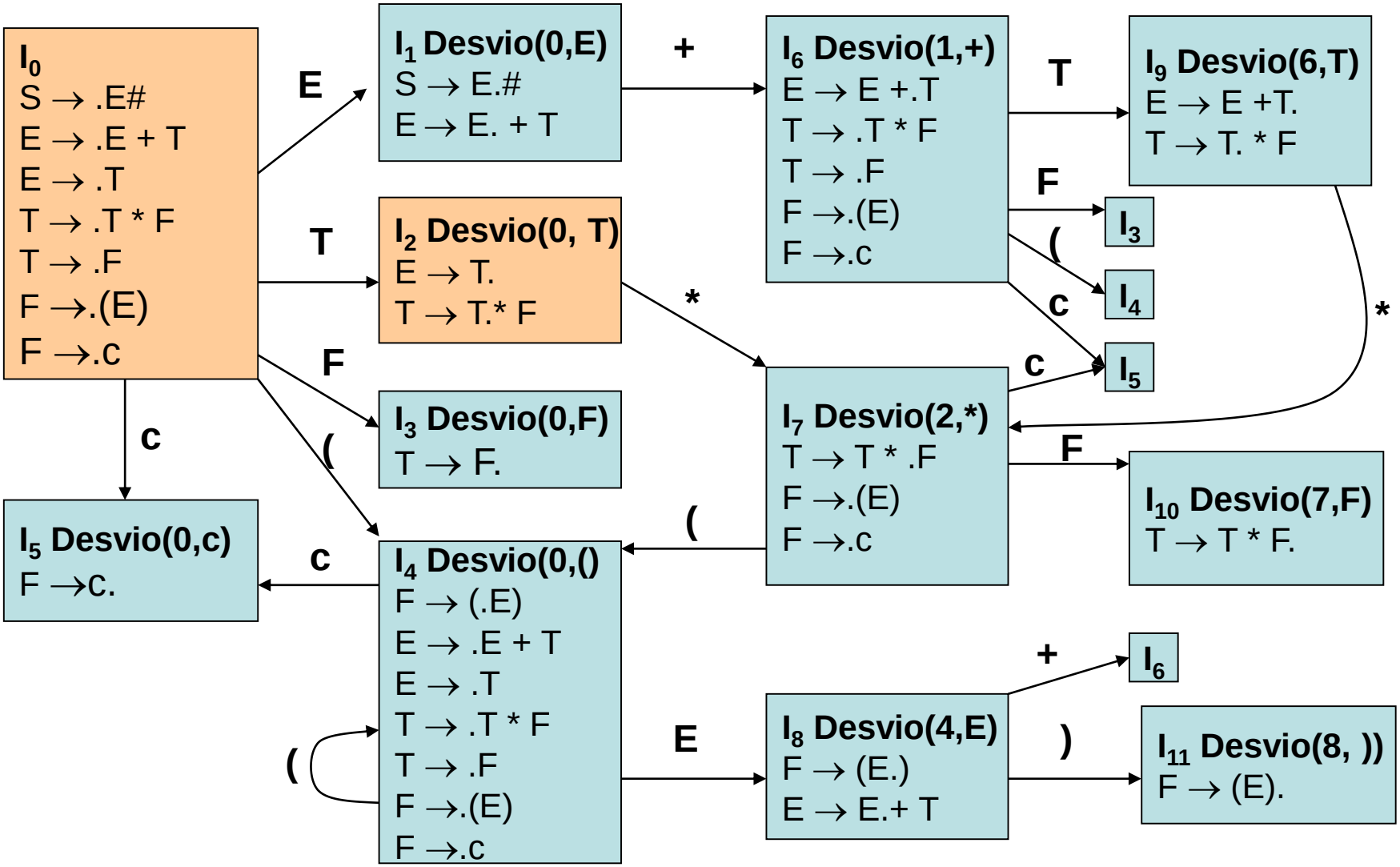
Entrada: c + c #



UDESC

Pilha : 0 T 2

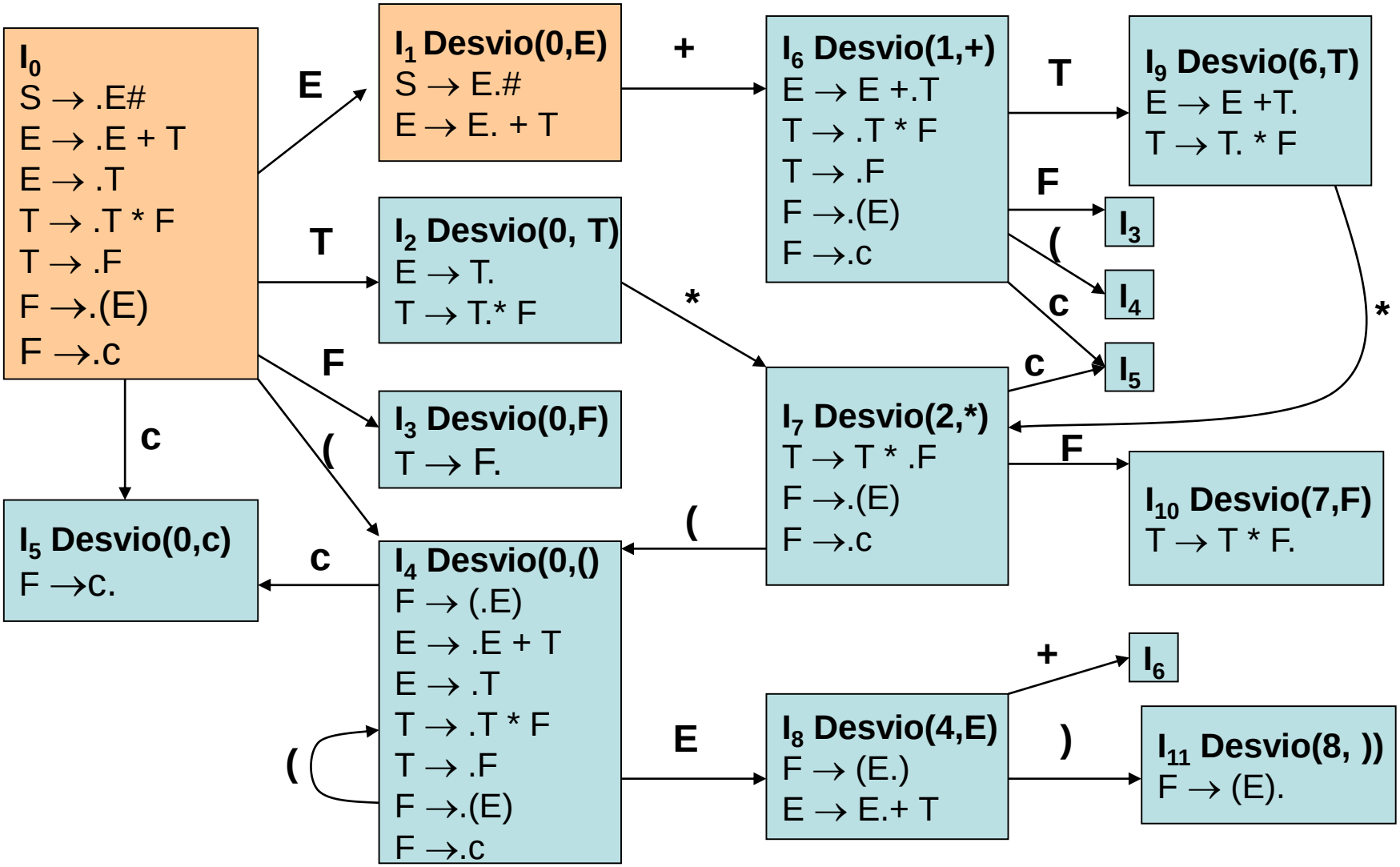
Entrada: c + c #



UDESC

Pilha : 0 E 1

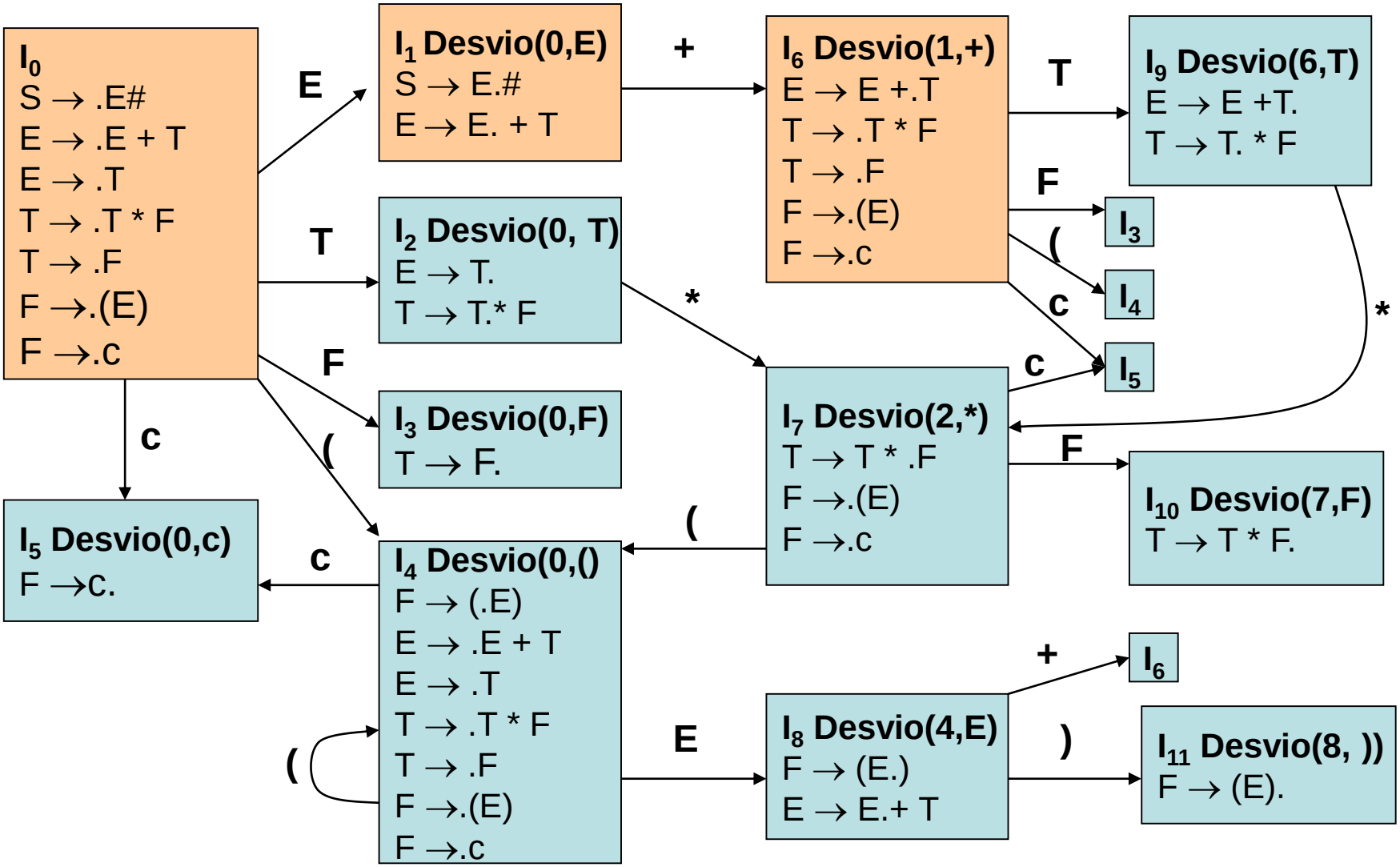
Entrada: c + c #



UDESC

Pilha : 0 E 1 + 6

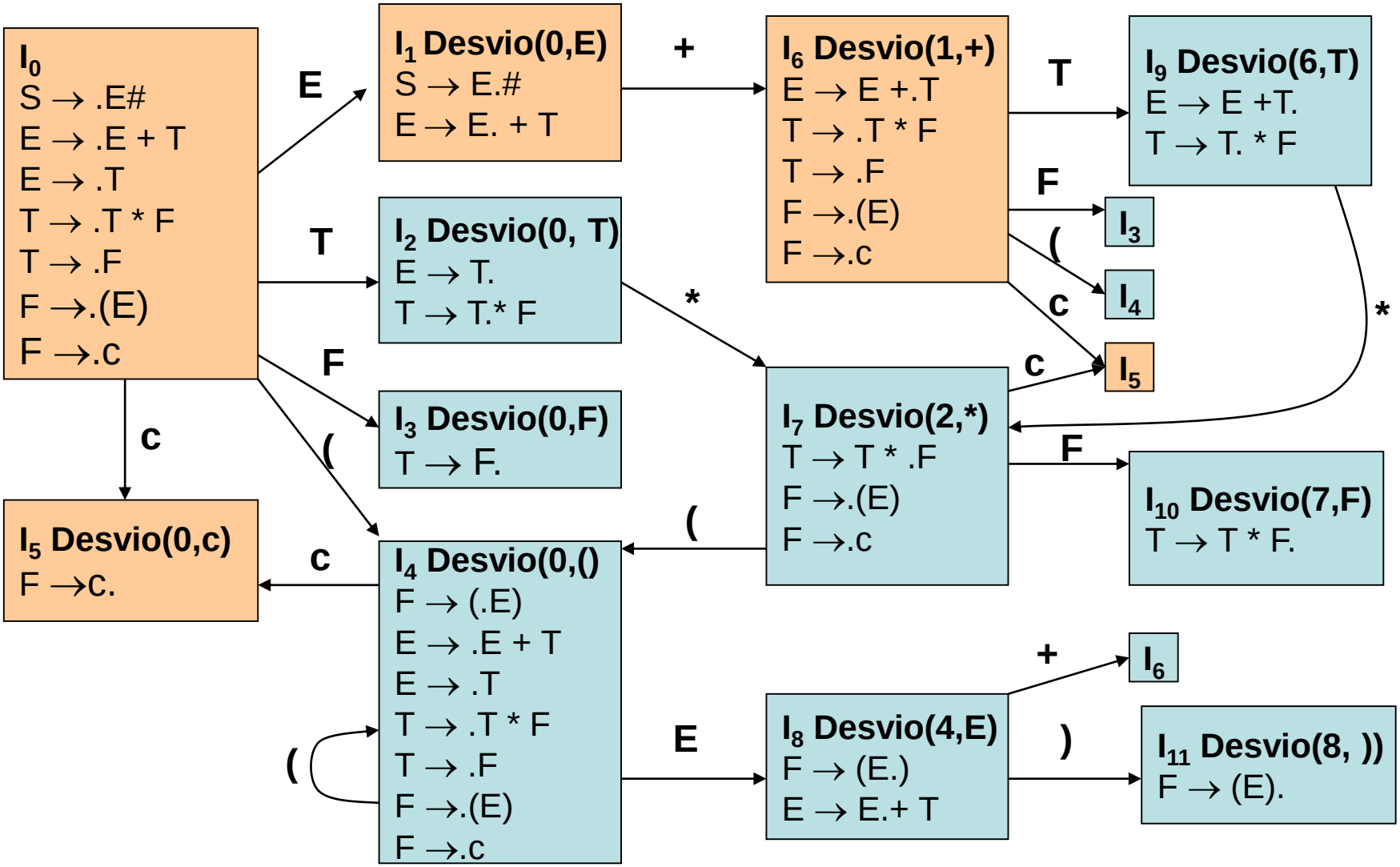
Entrada: c + c #



UDESC

Pilha : 0 E 1 + 6 c 5

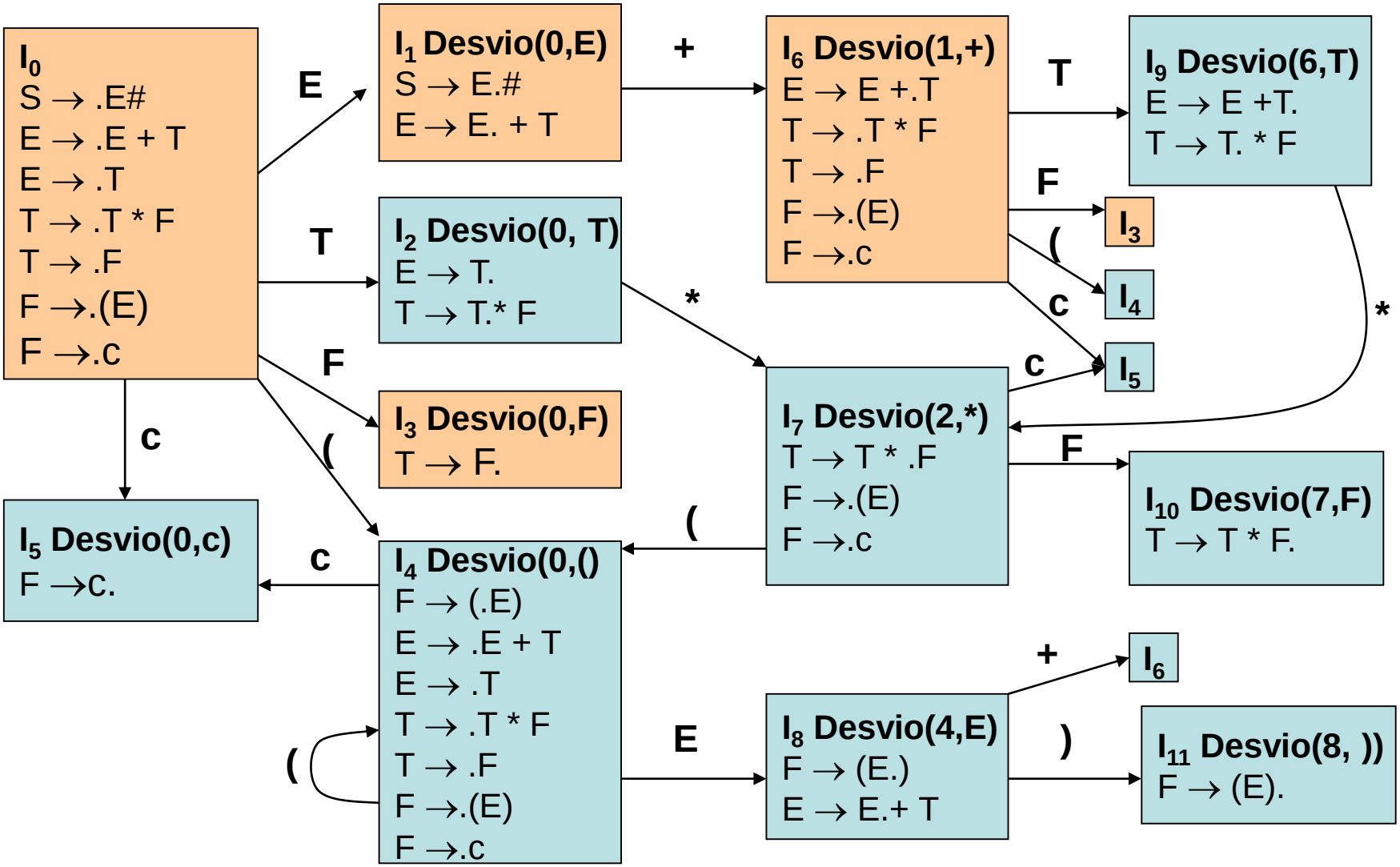
Entrada: c + c #



UDESC

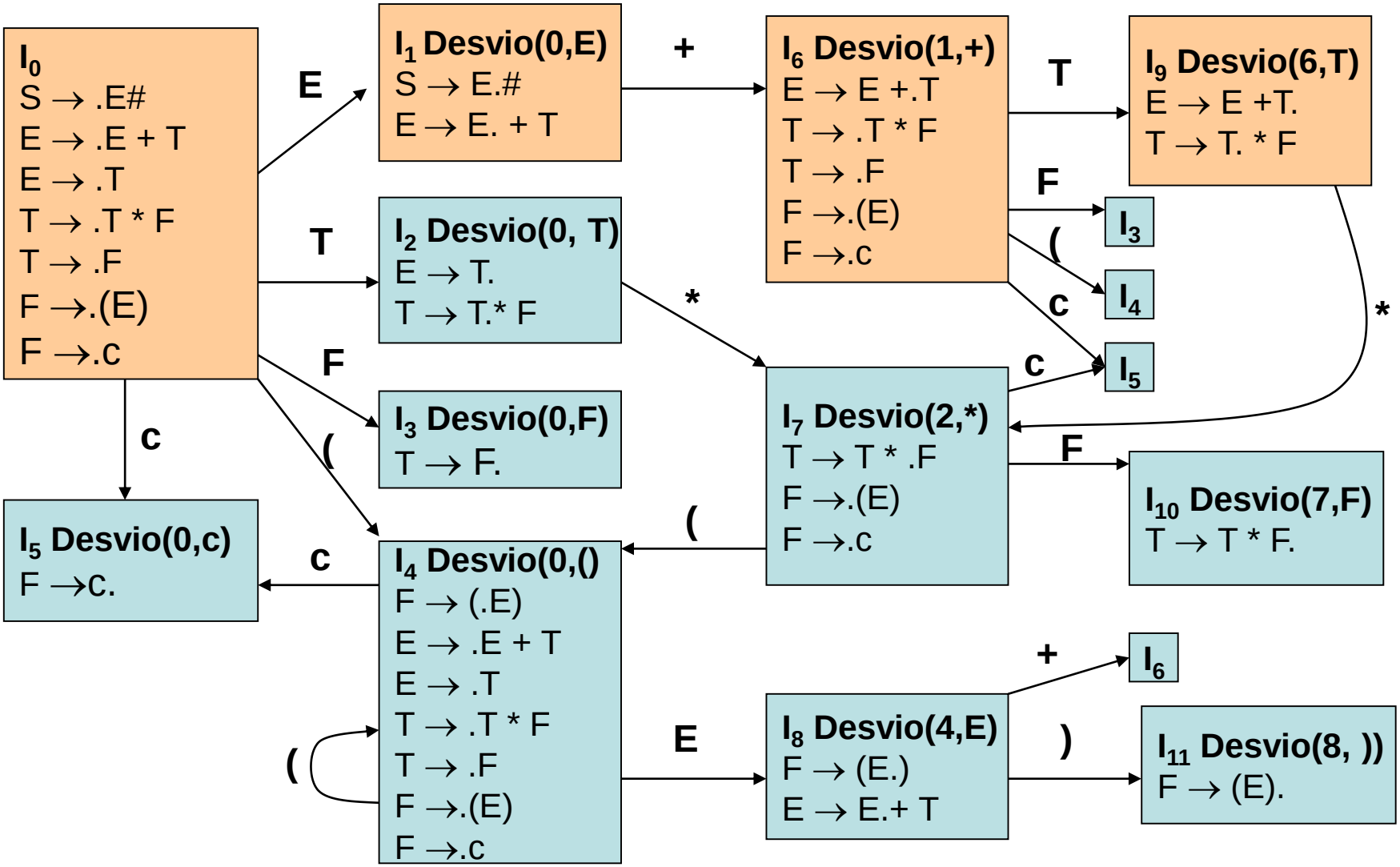
Pilha : 0 E 1 + 6 F 3

Entrada: c + c #



Pilha : 0 E 1 + 6 T 9

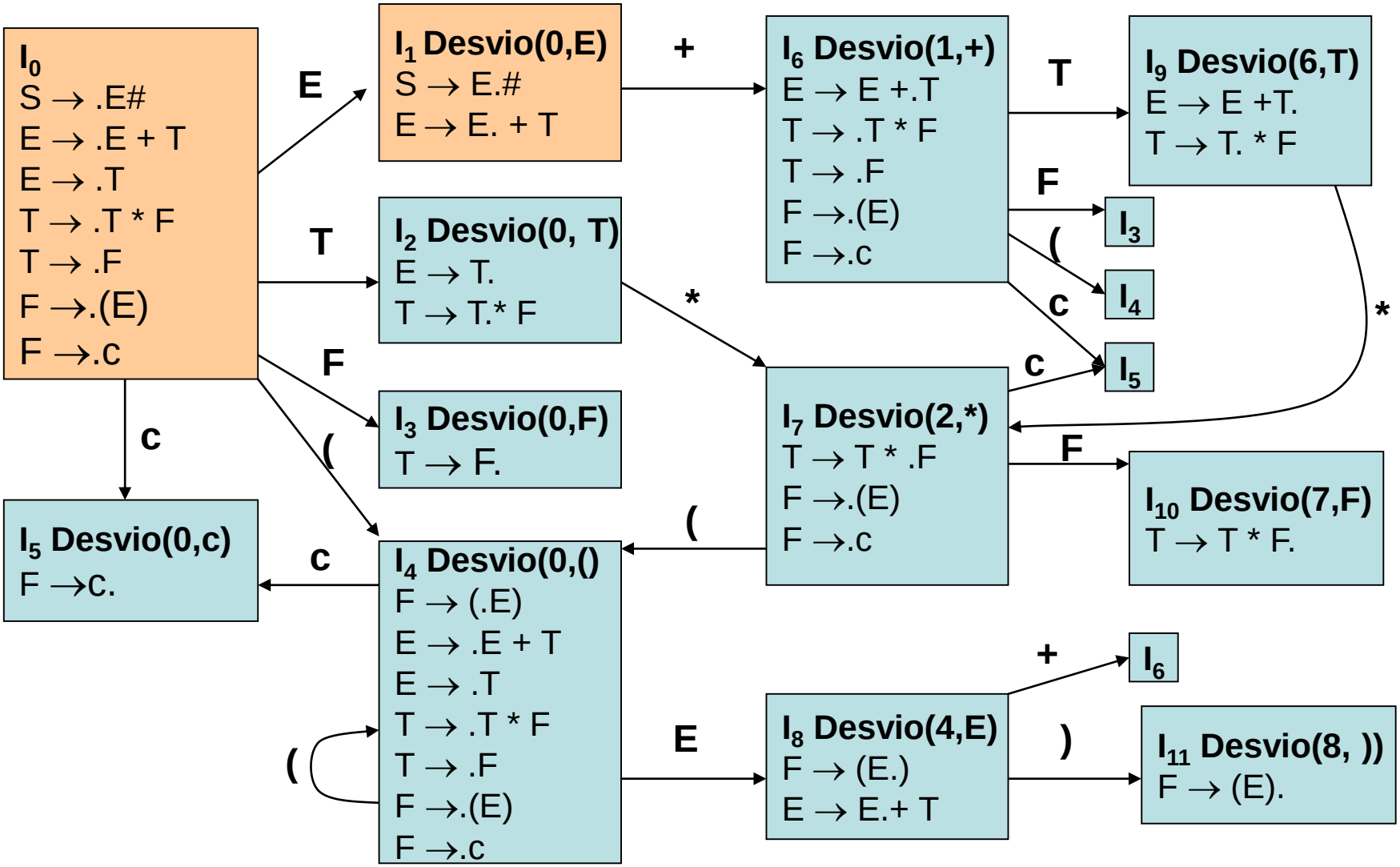
Entrada: c + c #



UDESC

Pilha : 0 E 1

Entrada: c + c #



UDESC

Tabelas SLR(1)

Um item LR(0), para uma gramática *G*, é uma produção de *G* com um ponto em alguma das posições do seu lado direito.

Exemplo: A produção E $\rightarrow$ E + T produz 4 itens:

[E $\rightarrow$ **.**E + T]

[E $\rightarrow$ E **.**+ T]

[E $\rightarrow$ E +**.**T]

[E $\rightarrow$ E + T**.**]

Intuitivamente o **.** indica até que parte da produção foi analisada em um determinado estado do analisador sintático.

Construção da Tabela Sintática SLR(1)

A construção da tabela sintática é auxiliada por duas operações:

Fechamento:

Sendo I um conjunto de itens da gramática, o fechamento de I é definido por duas regras:

1. Inicialmente cada item de *I* é adicionado em FECHAMENTO(*I*).
2. Se [$A \rightarrow \alpha.B\beta$] estiver em FECHAMENTO(*I*) e $B \rightarrow \gamma$ for uma produção, adicionar o item [$B \rightarrow .\gamma$] a FECHAMENTO(*I*). Essa regra é aplicada até que nenhum novo item possa ser adicionado.

Desvio:

A operação DESVIO(*I*, *X*) é definida como o fechamento do conjunto de todos os itens [$A \rightarrow \alpha X.\beta$] tais que [$A \rightarrow \alpha.X\beta$] esteja em *I*.

Construção da Tabela Sintática SLR(1)

Construção de um conjunto de itens LR(0):

C ← FECHAMENTO(*S'* → .*S*#) /* Onde *S* é o símbolo inicial da linguagem */
Repetir
    Para cada conjunto de itens *I* em *C* e cada símbolo gramatical *X*
        tal que DESVIO(*I*,*X*) não seja vazio e não esteja em *C*
            Incluir Desvio(*I*,*X*) em *C*
Até que não haja mais conjunto de itens a serem incluídos em *C*

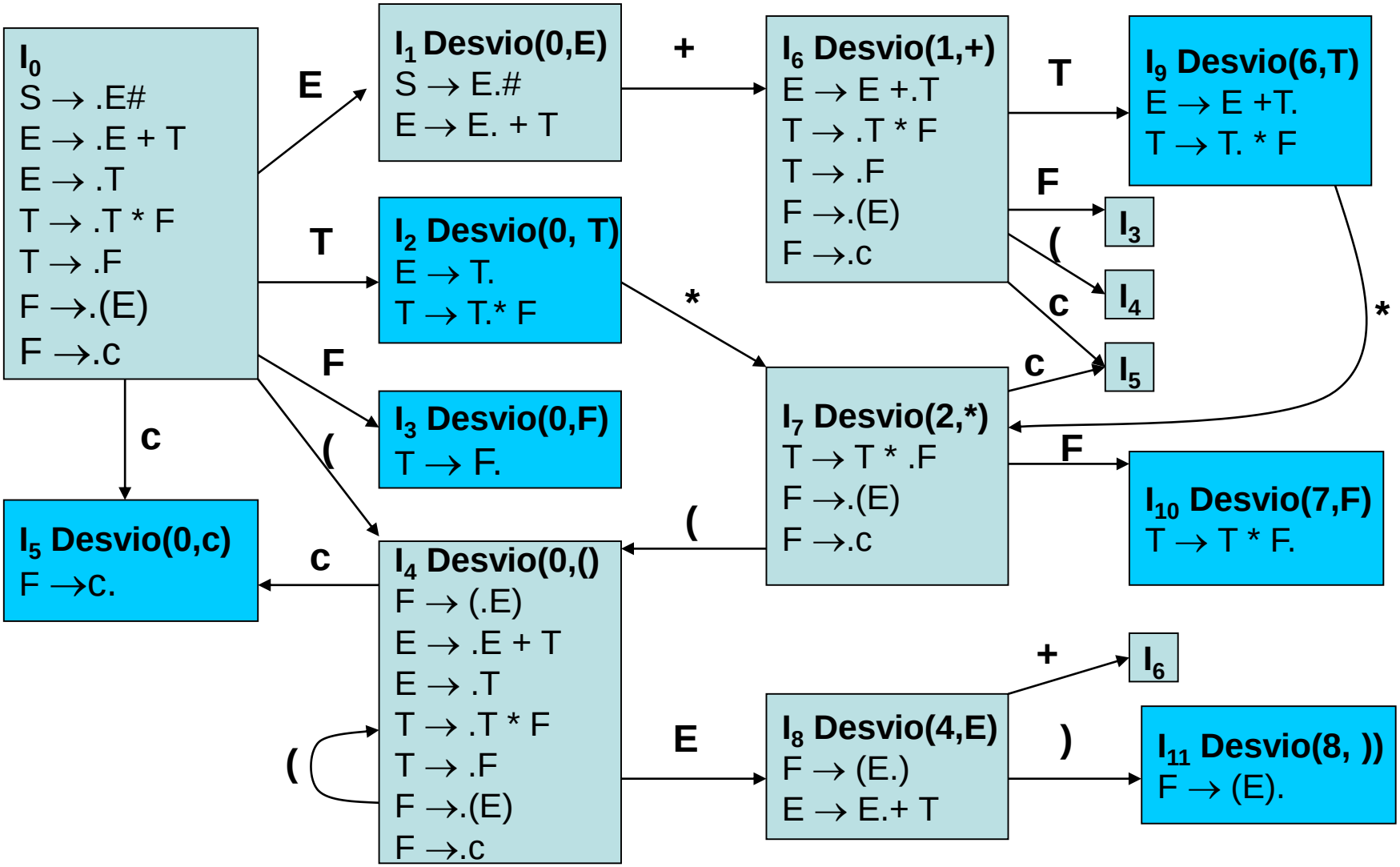
Construção da Tabela Sintática SLR(1)

Construção da tabela sintática SLR(1):

1. Construir o conjunto de itens $C = \{I_0, I_1, ..., I_n\}$
2. Cada estado i é construído a partir de I_i. As ações sintáticas são determinadas como:

- Se $[A \rightarrow \alpha.a\beta]$ estiver em I_i e DESVIO(I_i, a) = I_j então ação[i, a] = Empilhar j (se a for um terminal) ou desvio[i, a] = Empilhar j (se a for um não terminal)
- Se $[A \rightarrow \alpha.]$ estiver em I_i então ação(i, a) = reduzir através de $A \rightarrow \alpha$, para todo a em SEGUINTES(A).
- Se $[S' \rightarrow S.\#]$ estiver em I_i então ação(i, #) = aceitar

Construção da Tabela Sintática SLR(1)



Construção da Tabela Sintática SLR(1)

I_5 Desvio(0,c)
F →c.

SEGUINTES(F) = { +, *,), #
}

I_2 Desvio(0, T)
E → T.
T → T.* F

SEGUINTES(E) = { +,), # }

I_3 Desvio(0,F)
T → F.

SEGUINTES(T) = { +, *,), #
}

I_9 Desvio(6,T)
E → E +T.
T → T. * F

SEGUINTES(E) = { +,), # }

I_{10} Desvio(7,F)
T → T * F.

SEGUINTES(F) = { +, *,), #
}

I_{11} Desvio(8,))
F → (E).

SEGUINTES(F) = { +, *,), #
}

Se [$A \rightarrow \alpha.$] estiver em I_i então ação(i, a) = redução através de $A \rightarrow \alpha$, para todo a em SEGUINTES(A).

Construção da Tabela Sintática SLR(1)

No método SLR(1), como a decisão de reduzir aplicando uma produção $A \rightarrow \alpha$, é tomada usando o conjunto SEGUINTES(A) e não o contexto onde α ocorreu, algumas gramáticas LR(1) podem apresentar conflitos empilhar/reduzir se tentamos construir as tabelas usando esse método. Por o exemplo:

$S \rightarrow L = R$

$S \rightarrow R$

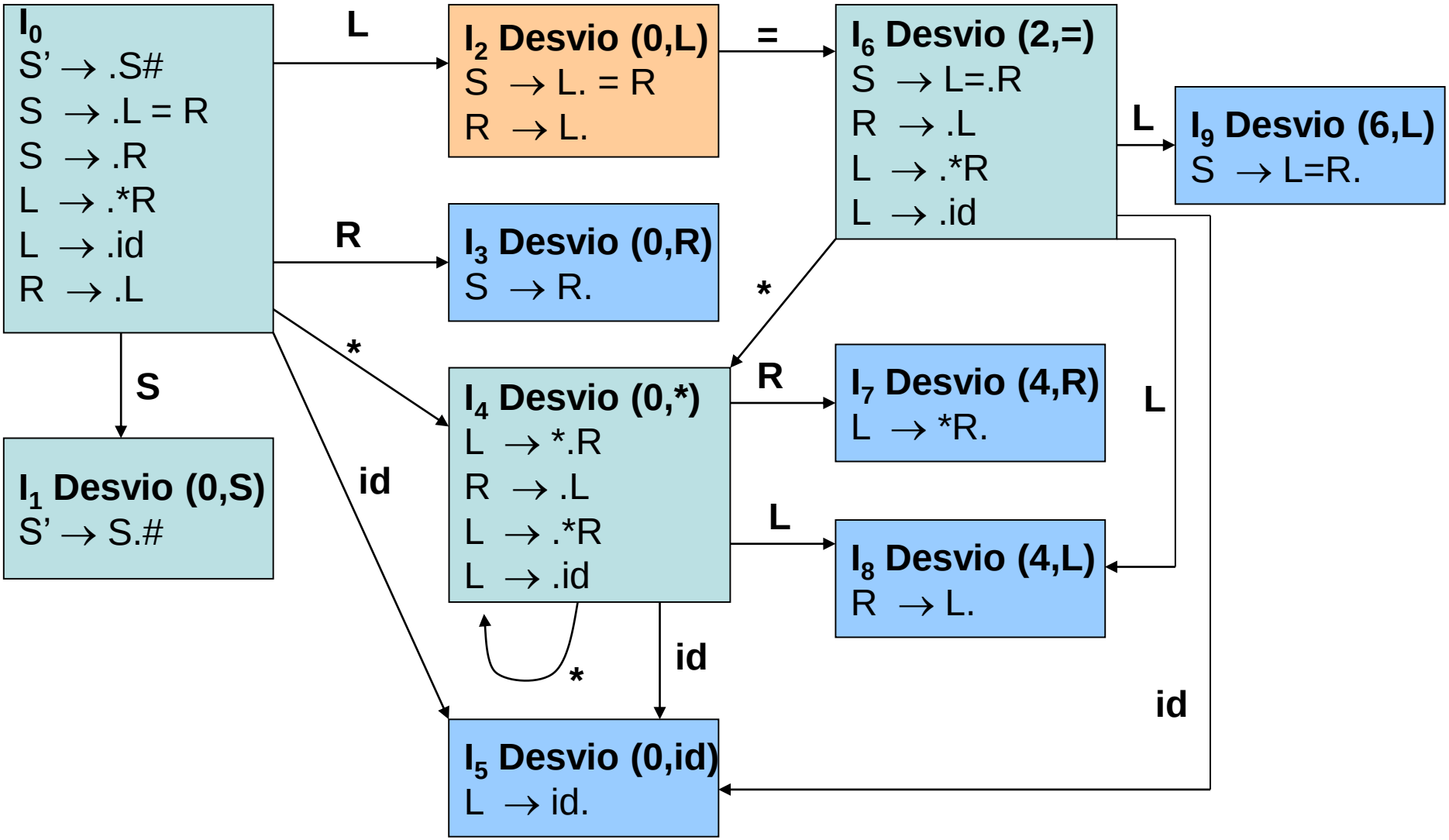
$L \rightarrow *R$

$L \rightarrow$ id

$R \rightarrow L$

*(Nesse exemplo os não terminais L e R representam **l-value** e **r-value** respectivamente)*

Construção da Tabela Sintática SLR(1)



Construção da Tabela Sintática SLR(1)

I_2 Desvio (0,L)
S $\rightarrow$ L. = R
R $\rightarrow$ L.

SEGUINTES(R) = {=, # }

No estado 2, a ação reduzir R $\rightarrow$ L deve ser executada para todos os seguintes de R, o que nesse caso ocasiona um conflito empilhar/reduzir. Entretanto não existe forma sentencial da gramática que inicie com **R =**. Para tratar essa gramática é necessário um método que carregue mais informação sobre o contexto para o estado.

Construção da Tabela Sintática LR(1)

Fechamento(*I*):

Repetir

Para cada item [$A \rightarrow \alpha.B\beta$, a] em I, cada produção $B \rightarrow \gamma$ na Gramática

e cada termina b em PRIMEIROS(βa) tal que [$B \rightarrow .\gamma$, b] não esta em I

Faça Incluir [$B \rightarrow .\gamma$, b] em I

até que não seja mais possível adicionar itens a I.

Desvio(*I*, *X*): é fechamento do conjunto de itens [$A \rightarrow \alpha X.\beta$, a] tais que

[$A \rightarrow \alpha.X\beta$, a] esta em I.

Itens(*G'*):

C = {FECHAMENTO({[$S' \rightarrow .S\#$, ε]})} (*Onde S é o símbolo inicial da gramática*)

Repetir

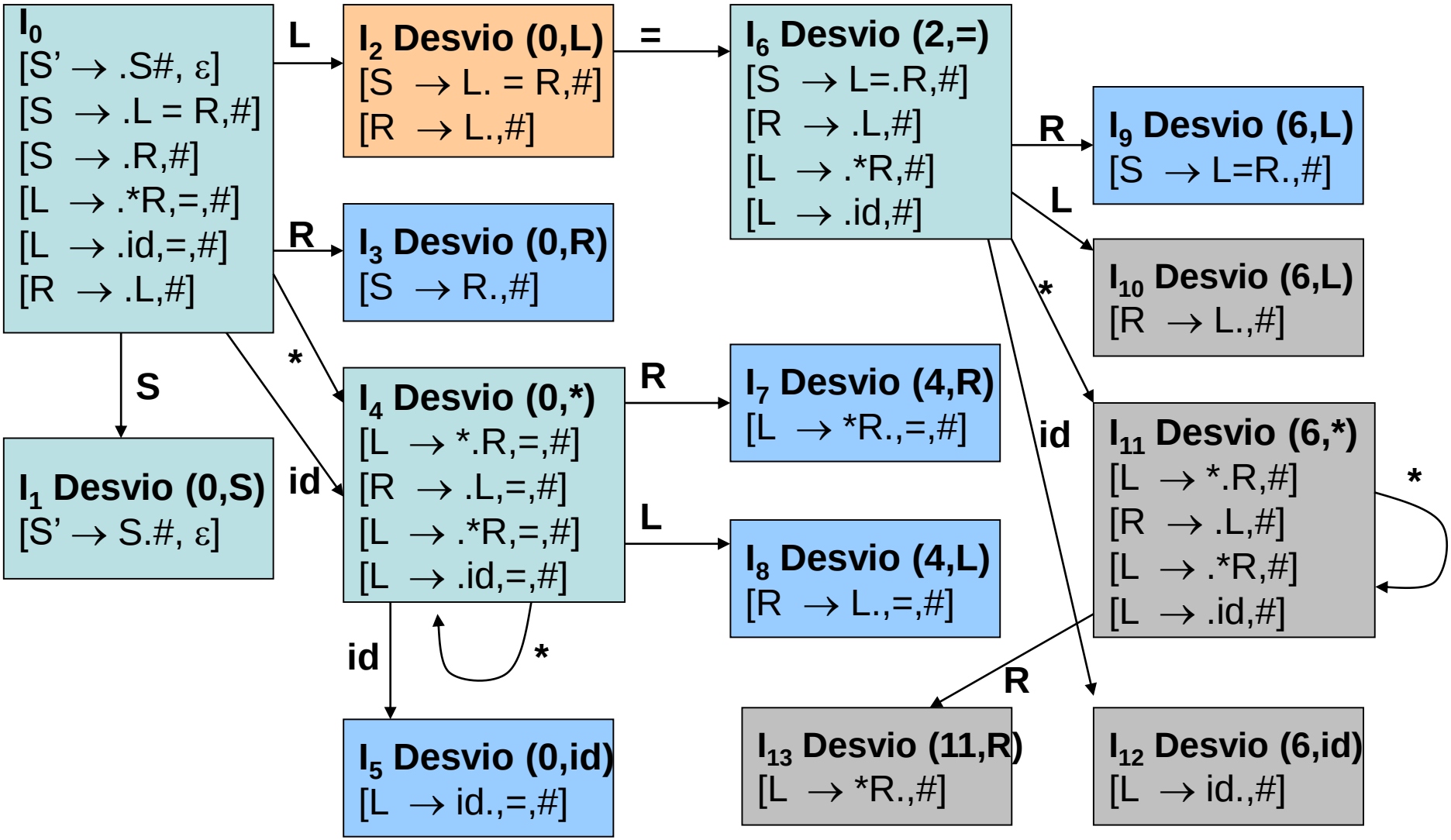
Para cada conjunto de itens I em C e cada símbolo gramatical X tal que

Desvio(I, X) $\neq \emptyset$ e Desvio(I, X) $\notin C$

Faça Incluir Desvio(I, X) em C

até que nenhum novo item possa ser adicionado a C

Construção da Tabela Sintática LR(1)



UDESC

LALR – lookahead LR

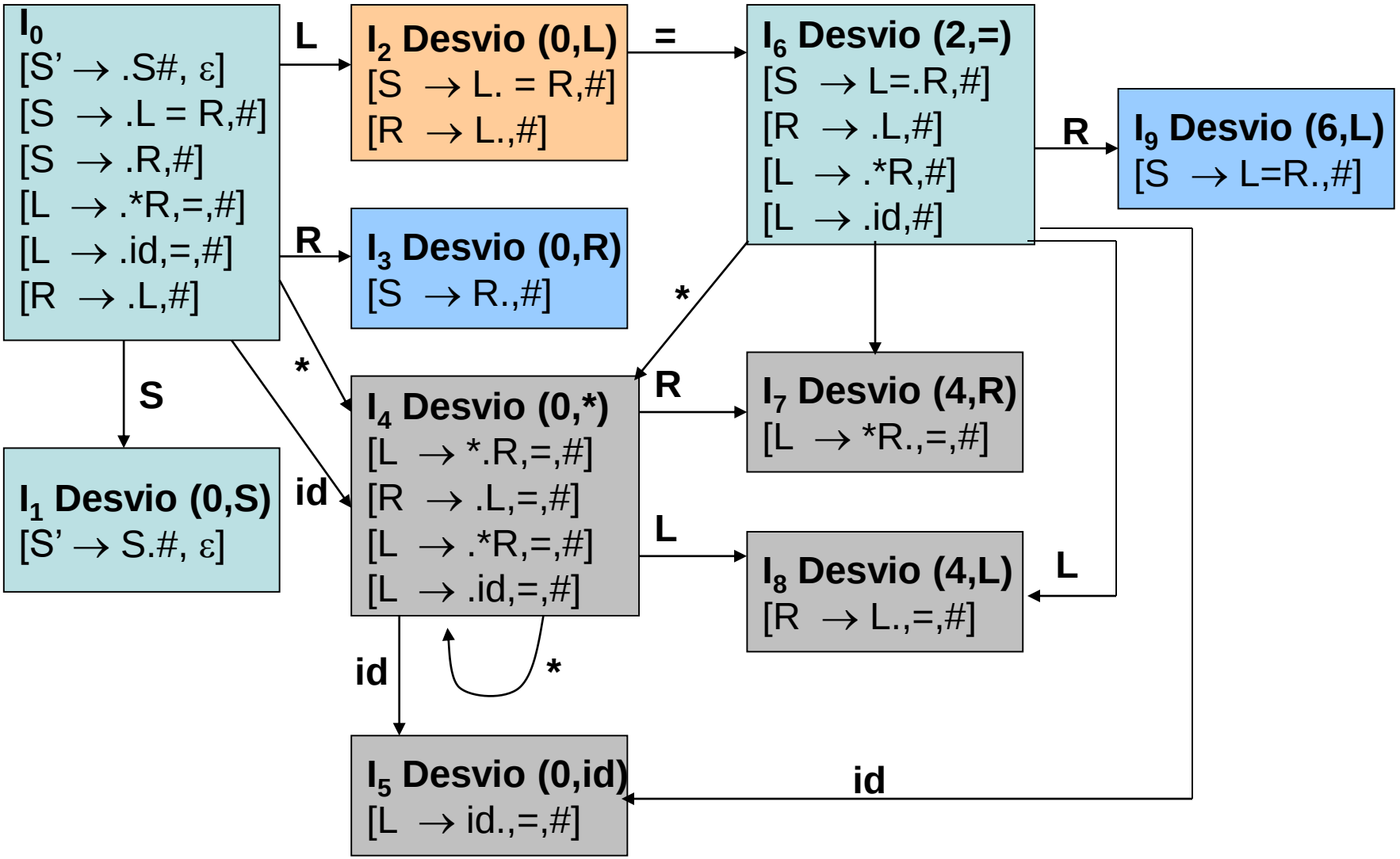
A idéia geral é construir o conjunto de itens LR(1) e, se nenhum conflito aparecer, combinar os itens com núcleo comum.

Algoritmos eficientes para a geração de tabelas LALR constroem o conjunto de itens LR(0) e numa segunda etapa determinam os *lookaheads* correspondentes de cada item.

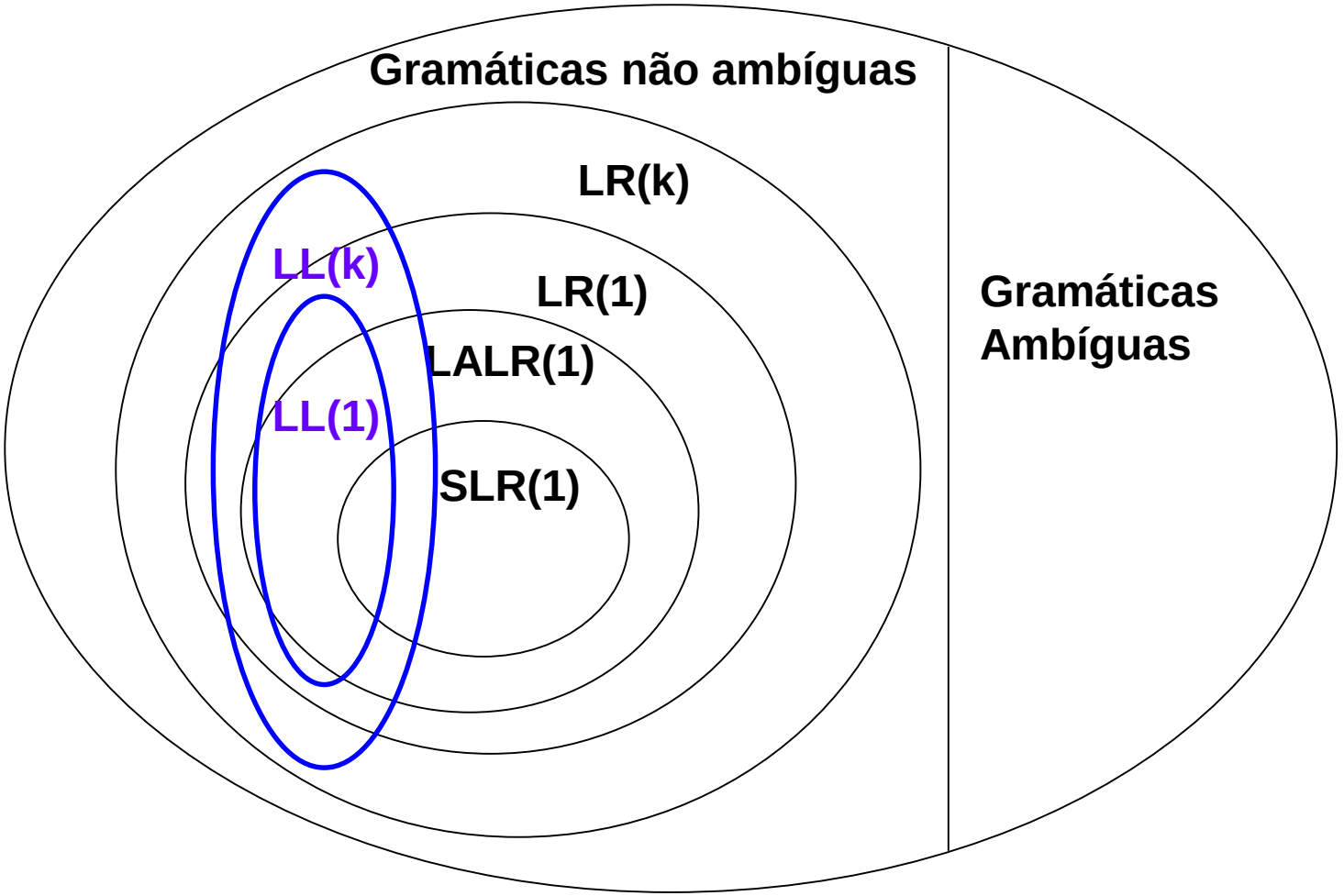
O método LALR:

- Trata a maioria dos casos presentes em linguagens de programação;
- Na grande maioria dos casos o número de estados é muito inferior ao número de estados gerados pelo método LR(1).
- Comparando com o método LR(1), em algumas situações, a detecção de erro é postergada, reduções desnecessárias são aplicadas antes que o erro seja detectado.

Construção da Tabela Sintática LALR(1)



Hierarquia de Gramáticas Livres de Contexto

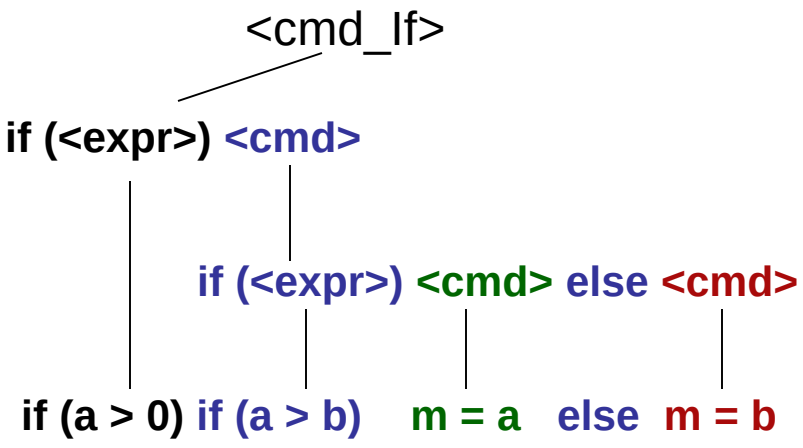


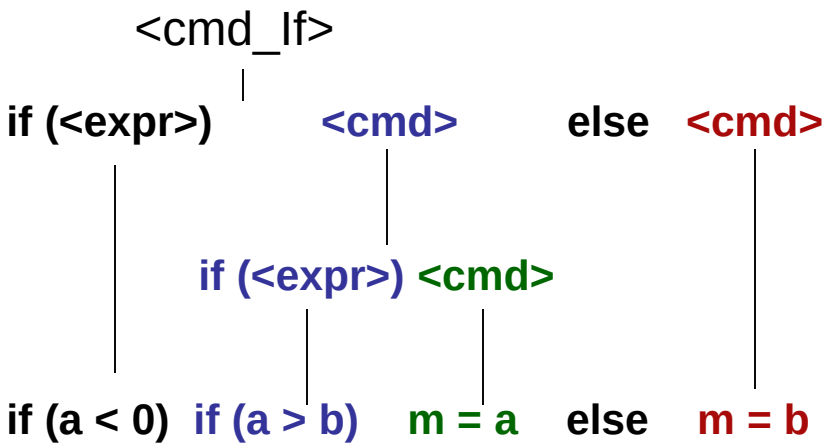
Uso de Gramáticas Ambíguas

<cmd_If> → if (<expr>) <cmd> else <cmd>

| if (<expr>) <cmd>

<cmd> → ... | <cmd> | ...





*Essa gramática é ambígua, como consequência temos um conflito **empilhar/reduzir** no estado em que ocorre a transição para o token **"else"**. Caso esse conflito seja resolvido escolhendo a opção **empilhar** as reduções executadas serão as correspondentes a primeira árvore.*

UDESC

Lex

```
delim       [ \t]
ws          {delim}+
digito      [0-9]
num         {digito}+(\.{digito}*(E[+-]?{digito}+)?)?

%%
{ws}        {}
"+" {return TADD;}
"-"  {return TSUB;}
"*"  {return TMUL;}
"/"  {return TDIV;}
"("  {return TAPAR;}
")"  {return TFPAR;}
\n   {return TFIM;}
{num}       {yylval=atof(yytext); return TNUM;}
```

UDESC

yacc

```
%{
#include <stdio.h>
#include <stdlib.h>
#define YYSTYPE double
%}

%token TADD TMUL TSUB TDIV TAPAR TFPAR TNUM TFIM

%%
```

UDESC

Yacc/Bison

```
Linha :Expr TFIM {printf("Resultado:%lf\n", $1);exit(0);}
      ;
Expr: Expr TADD Termo {$$ = $1 + $3;}
      | Expr TSUB Termo {$$ = $1 - $3;}
      | Termo
      ;
Termo: Termo TMUL Fator {$$ = $1 * $3;}
      | Termo TDIV Fator {$$ = $1 / $3;}
      | Fator
      ;
Fator: TNUM
      | TAPAR Expr TFPAR {$$ = $2;}
      ;
%%
```

UDESC

Yacc/Bison

```
int yyerror (char *str)
{
    printf("%s - antes %s\n", str, yytext);

}

int yywrap()
{
    return 1;
}
```

UDESC

Programa

```
#include <stdio.h>

extern FILE *yyin;

int main()
{
    yyin = stdin;
    printf("Digite uma expressão:");
    yyparse();
    return 0;
}
```

UDESC

Definição Dirigida pela Sintaxe

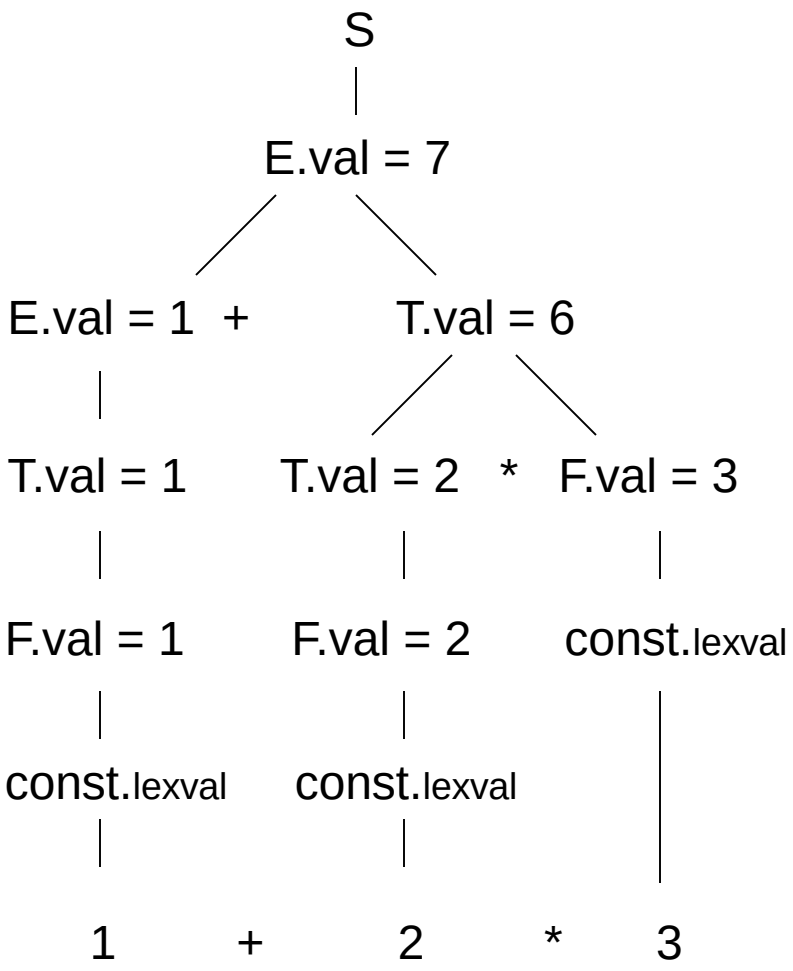
É uma gramática livre de contexto na qual a cada símbolo é associado um conjunto de atributos. A cada produção pode estar associado um conjunto de regras semânticas. Essa regras podem alterar valores de atributos, emitir código, atualizar a tabela de símbolos, emitir mensagens de erro ou realizar quaisquer outras atividades. Em geradores de analisadores sintáticos essas regras são geralmente descritas em uma linguagem de programação.

Os atributos são classificados como:

- **Atributos Sintetizados**: O valor do atributo de um nó é computado a partir dos atributos de seus filhos.
- **Atributos Herdados:** O valor do atributo de um nó é computado a partir dos atributos dos nós irmãos e/ou pais.

Definição Dirigida pela Sintaxe

Produção	Regra Semântica
$S \rightarrow E$	{Imprimir (E.val)}
$E \rightarrow E_1 + T$	{E.val = E_1.val + T.val}
$E \rightarrow T$	{E.val = T.val}
$T \rightarrow T_1 * F$	{T.val = T_1.val * F.val}
$T \rightarrow F$	{T.val = F.val}
$F \rightarrow (E)$	{F.val = E.val}
$F \rightarrow$ ***const***	{F.val = ***const***.*lexval*}



Árvore Sintática Abstrata

$E \rightarrow E_1 + T$ {E.ptr = criarNo (‘+’, E_1.ptr, T.ptr)}

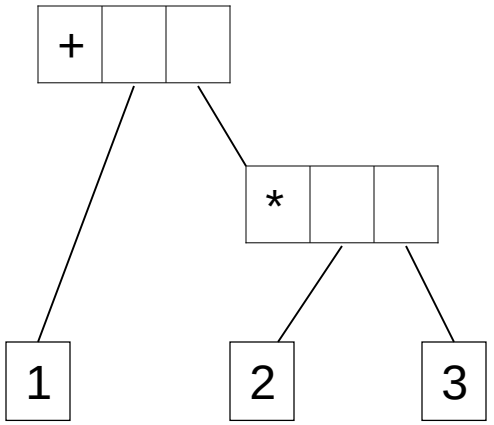
$E \rightarrow T$ {E.ptr = T.ptr}

$T \rightarrow T_1 * F$ {T.ptr = criarNo (‘*’, T_1.ptr, F.ptr)}

$T \rightarrow F$ {T.ptr = F.ptr}

$F \rightarrow (E)$ {F.ptr = E.ptr}

$F \rightarrow$ ***const*** {F.ptr = criarFolha(***const.lexval***)}

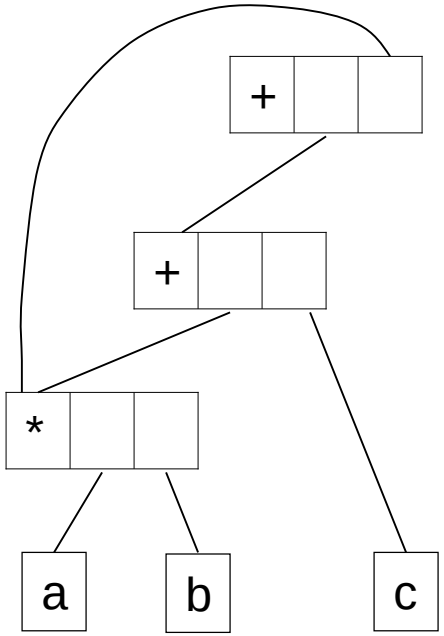


1 + 2 * 3

DAG - Grafo Direcionado Acíclico

Identifica as subexpressões comuns. Exemplo:

DAG que representa a expressão: *a* * *b* + *c* + *a* **b*.



UDESC

Código de 3 endereços

Uma sequência de enunciados na forma:

$x = y$ ***op*** z

Sendo *x*, *y* e *z* nomes, constantes ou dados temporários (criados pelo compilador) e ***op*** o código que representa uma operação qualquer.

Uma versão linearizada da árvore sintática, é assim chamado por cada instrução poder conter até três endereços, dois para os operandos e um para o resultado. Bastante semelhante a uma linguagem de montagem.

UDESC

Código de 3 endereços

Exemplo: a = 2 * b + c

t1 = 2 * b

t2 = t1 + c

a = t2

Código de 3 endereços

$S \rightarrow$ id = E {S.cod = E.cod ++ gerar(id.lexval = E.local)}

$E \rightarrow E_1$ + T {E.local = novoTemporario();
E.cod = E_1.cod ++T.cod ++ gerar(E.local = E_1.local + T.local)}

$E \rightarrow$ T {E.local = T.local; E.cod = T.cod}

$T \rightarrow T_1$ * F {T.local = novoTemporario();
T.cod = T_1.cod ++ F.cod ++ gerar(T.local = T_1.local * F.local)}

$T \rightarrow$ F {T.local = F.local; T.cod = F.cod}

$F \rightarrow$ (E) {F.local = E.local; F.cod = E.cod}

$F \rightarrow$ id {F.local = id.lexval; F.cod =“”}

$F \rightarrow$ const {F.local = const.lexval; F.cod =“”}

UDESC

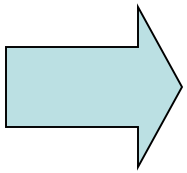
Código de 3 endereços

Alguns enunciados comumente usados:

- x = y op z, onde op é uma operação binária.
- x = op y, onde op é uma operação unária.
- x = y, enunciado de cópia.
- goto L, desvio incondicional.
- if x relop y goto L, onde relop é um operador relacional.
- param x, passagem de parâmetro para funções/procedimentos.
- call p, chamada de uma função/procedimento.
- x = a[i] ou a[i] = x, atribuições indexadas.
- *x = y ou x = *y, indireções.

Código de 3 endereços

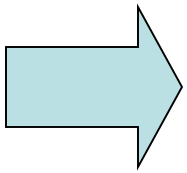
```
n = 1;
f = 1;
while (n < 10)
{
    f = f * n;
    n = n + 1;
}
```



```
      n = 1
      f = 1
L1:   if n < 10 goto L2
      goto L3
L2:   f = f * n
      n = n + 1
      goto L1:
L3:
```

Código de 3 endereços

```
n = 1;
f = 1;
while (n < 10)
{
    f = f * n;
    n = n + 1;
}
```

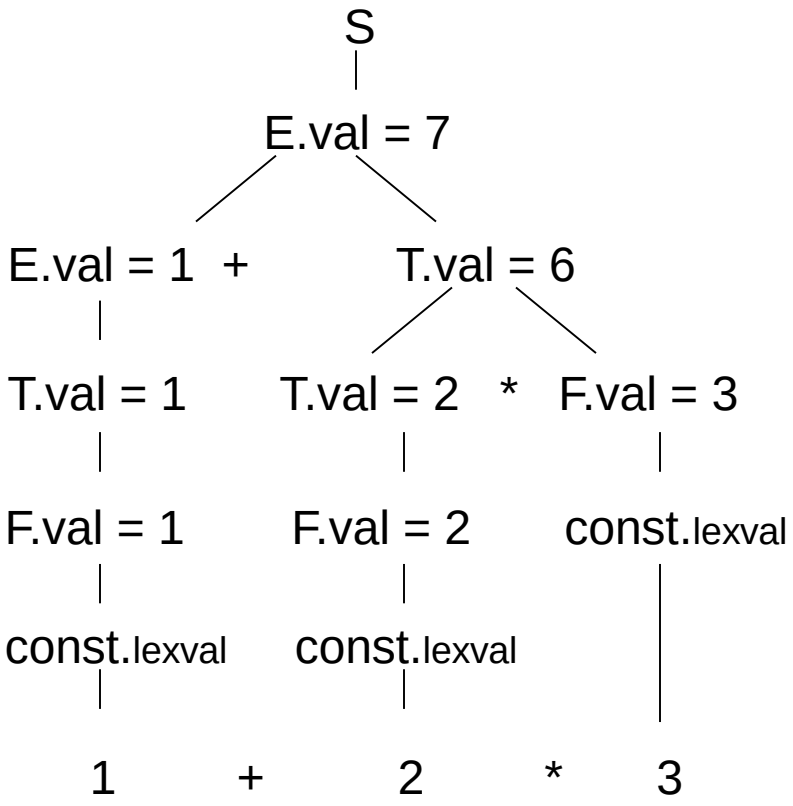


```
    n = 1
    f = 1
L1: if n >= 10 goto L3
    f = f * n
    n = n + 1
    goto L1:
L3:
```

Definições S-atribuídas

Definições dirigidas pela sintaxe que possuem apenas atributos sintetizados.

Produção	Regra Semântica
$S \rightarrow E$	{Imprimir (E.val)}
$E \rightarrow E_1 + T$	{E.val = E_1.val + T.val}
$E \rightarrow T$	{E.val = T.val}
$T \rightarrow T_1 * F$	{T.val = T_1.val * F.val}
$T \rightarrow F$	{T.val = F.val}
$F \rightarrow (E)$	{F.val = E.val}
$F \rightarrow$ ***const***	{F.val = ***const****.lexval*}



UDESC

Definições L-atribuídas

Uma definição dirigida pela sintaxe é *L-atribuída* se cada atributo herdado de X_j, $1 \leq j \leq n$, do lado direito de uma produção, $A \rightarrow X_1X_2...X_n$ depende somente:

1. Dos atributos dos símbolos $X_1X_2...X_{j-1}$ (símbolos a esquerda de X_j).
2. Dos atributos herdados de A.

UDESC

Tradução *Top-Down*

$S \rightarrow E$ {imprimir (E.val)}

$E \rightarrow T$ {E'.h = T.val} E' {E.val = E'.s}

$E' \rightarrow$ **+**T {E'_1.h = E'.h + T.val} E_1 {E'.s = E'_1.s}

$E' \rightarrow \varepsilon$ {E'.s = E'.h}

$T \rightarrow F$ {T'.h = F.val} T' {T.val = T'.s}

$T' \rightarrow$ * F{T'_1.h = T'.h * F.val} T' {T'.s = T_1'.s}

$T' \rightarrow \varepsilon$ {T'.s = T'.h}

$F \rightarrow$ ***const*** {F.val = ***const***.lexval}

$F \rightarrow$**(**{push(T'.h); push(E'.h)} E {E'.h = pop(); T'.h = pop()}**)**

UDESC

Análise Semântica

(Verificação de Contexto)

D $\rightarrow$ **var** S

S $\rightarrow$ **id** L {atribuirTipo(id.lexval, L.tipo)}

S $\rightarrow$ S **id** L {atribuirTipo(id.lexval, L.tipo)}

L $\rightarrow$ **,** **id** L_1 {atribuirTipo(id.lexval, L.tipo); L.tipo = L_1.tipo}

L $\rightarrow$ **:**T {L.tipo = T.tipo}

T $\rightarrow$ **integer** {T.tipo = integer}

T $\rightarrow$ **string** {T.tipo = string}

UDESC

Análise Semântica
(Verificação de Contexto)

$E \rightarrow E_1$ **+** T {if (E_1.tipo = T.tipo) then E.tipo = E_1.tipo else error()}

$E \rightarrow$ T {E.tipo = T.tipo}

$T \rightarrow T_1$ ***** F {if (T_1.tipo = F.tipo) then T.tipo = T_1.tipo else error()}

$T \rightarrow$ F {T.tipo = F.tipo}

$F \rightarrow$ **id** {F.tipo = consultaTipo(id.lexval);}

$F \rightarrow$ **constInt** {F.tipo = Inteiro}

$F \rightarrow$ **constReal** {F.tipo = Real}

Obs: Em uma situação real as regras semânticas devem implementar a coerção dos tipos.

Expressões Lógicas e Relacionais

B $\rightarrow$ B_1 **or** M C {corrigir(B_1.listaf, M.label);
B.listav = merge(B_1.listav, C.listav);
B.listaf = C.listaf;}

B $\rightarrow$ B_1 **and** M C {corrigir(B_1.listav, M.label);
B.listaf = merge(B_1.listaf, C.listaf);
B.listav = C.listav;}

B $\rightarrow$ C {B.listav = C.listav; B.listaf = C.listaf;}

C $\rightarrow$ **not** C_1 {C.listav = C_1.listaf; C.listaf = C_1.listav;}

C $\rightarrow$ **(**B**)** {C.listav = B.listav; C.listaf = B.listaf;}

C $\rightarrow$ E_1 **rel** E_2 {C.listav = criaLista(proxInst);
C.listf = criaLista(proxInst+1);}
gerar(**if** E_1.local **rel** E_2.local **goto** _);
gerar (**goto** _);

M $\rightarrow$ ε {M.label = novolabel()}

Comandos de Seleção e Repetição

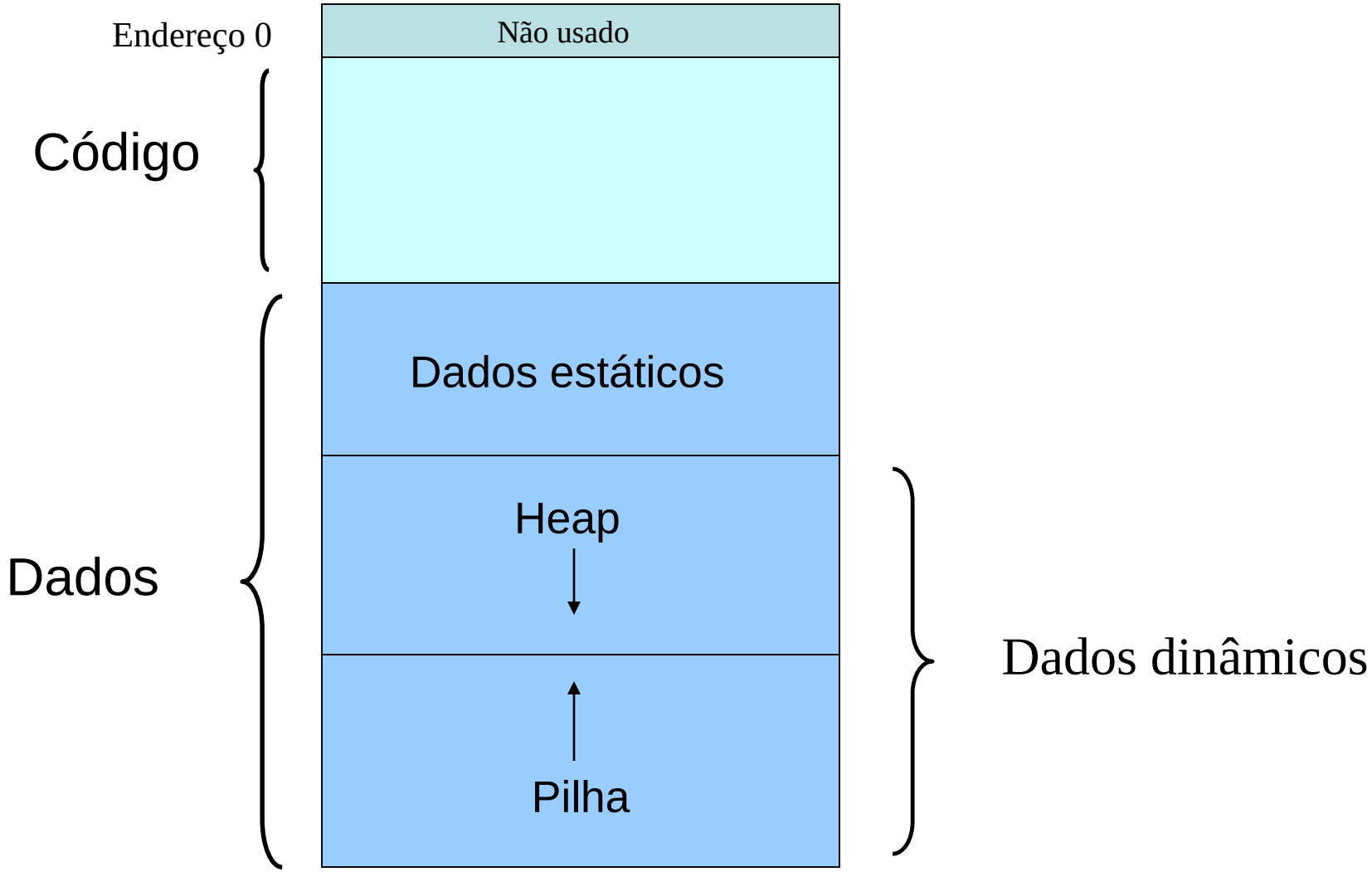
S → **if (**B**) then** M S {corrigir (B.listav, M.label);
corrigir(B.listaf, novolabel();}

S → **if (**B**) then** M_1 S N **else** M_2 S
{corrigir(B.listav, M_1.label);
corrigir(B.listf, M_2.label);
corrigir(N.listav, novoLabel();}

S → **while** M_1 **(**B**)** M_2 S
{corrigir(B.listav, M_2.label);
gerar(goto M_1.label);
corrigir(B.listaf, novolabel();}

N → ε {N.listav = criarLista(ProxInstr); gerar(goto _);}

Organização da Memória



UDESC

Dados Estáticos

A área de memória é reservada no início da execução do programa e liberada apenas no fim de sua execução (*e.g.* variáveis globais e variáveis locais declaradas com o modificador *static* em linguagem C).

UDESC

Dados Dinâmicos

Pilha – Área de armazenamento temporário onde é armazenado o registro de ativação das funções.

Heap – Área reservada para alocação dinâmica, permite ao programador alocar e liberar espaços de memória quando necessário (*e.g.* áreas de memória reservadas pelas funções *malloc* e liberadas pela função *free* em linguagem C).

UDESC

Registro de Ativação

As informações necessárias para execução de uma função/procedimento são gerenciadas utilizando um bloco de memória chamado registro de ativação, fazem parte do registro de ativação: parâmetros, endereço de retorno e variáveis locais.

Chamadas de Funções/Procedimentos

```
int fat (int n)
{
    if (n <= 1)
        return 1;
    return n * fat(n-1);
}
```

```
fat:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ebx
    cmpl $1, %ebx
    jl L1:
    movl $1, %eax
    movl %ebp, %esp
    popl %ebp
    ret
L1: subl $1, %ebx
    pushl %ebx
    call fat
    subl $4, %esp
    movl 8(%ebp), %ebx
    imull %ebx, %eax
    movl %ebp, %esp
    popl %ebp
    ret
```

```
fat:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ebx
    cmpl $1, %ebx
    jl L1:
    movl $1, %eax
    jmp L3
L1: subl $1, %ebx
    pushl %ebx
    call fat
    movl 8(%ebp), %ebx
    imull %ebx, %eax
L3: movl %ebp, %esp
    popl %ebp
    ret
```

Chamadas de Funções/Procedimentos

```
int fat (int n)
{
    if (n >= 1)
        return 1;
    return n * fat(n-1);
}
Int main()
{
    int x;
    x = fat(3);
}
```