Curso: Engenharia de Computação

Linguagens Formais e Compiladores

Prof. Clayton J A Silva, MSc
clayton.silva@professores.ibmec.edu.br

Análise Sintática

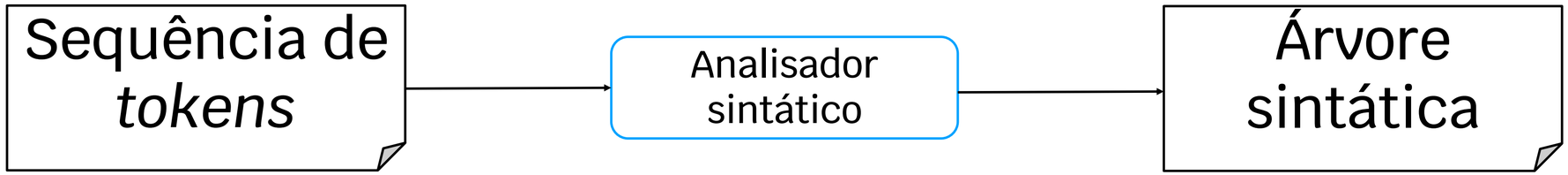
Introdução

- Determina a sintaxe, ou estrutura, de um programa
- A sintaxe de uma LP é definida normalmente pelas **regras gramaticais** de uma **gramática livre de contexto**
- As regras de uma gramática livre de contexto são **recursivas**
- Utiliza como estrutura básica algum tipo de árvore, denominada **árvore de análise sintática** ou simplesmente **árvore sintática**

Introdução

- Existem duas categorias gerais de algoritmos para análise sintática, conforme é construída a análise sintática:
  - **ascendente**
  - **descendente**

Processo de análise sintática



- Em geral, a sequência de tokens não é um parâmetro explícito de entrada, mas ativado pelo analisador por procedimento de varredura, como ***getToken()***

- A etapa de análise sintática do compilador pode se resumir à sua ativação, que retorna o resultado à árvore sintática, por exemplo,

  arvoreSintatica = analisar()

Processo de análise sintática

- Em geral, a estrutura da árvore é definida como uma **estrutura de dados dinâmica**, em que **cada nó é composto por um registro** cujos campos contêm os atributos requeridos para o restante do processo de compilação

- A economia de espaço de armazenamento é sempre levada em consideração em proveito do aumento de desempenho

- Tratamento de erros: mais complexo do que a varredura, pois (i) precisa **recuperar** a situação sem erros e **seguir com a análise** para descobrir outros erros; (ii) e pode precisar efetuar a **correção de erros**

Diagramas sintáticos

- Representações gráficas visuais para regras de BNF estendida.
- Elementos:
  - círculos ou formas ovais indicam os terminais
  - quadrados ou retângulos indicam não terminais
  - linhas direcionadas representam sequências

Árvores binárias

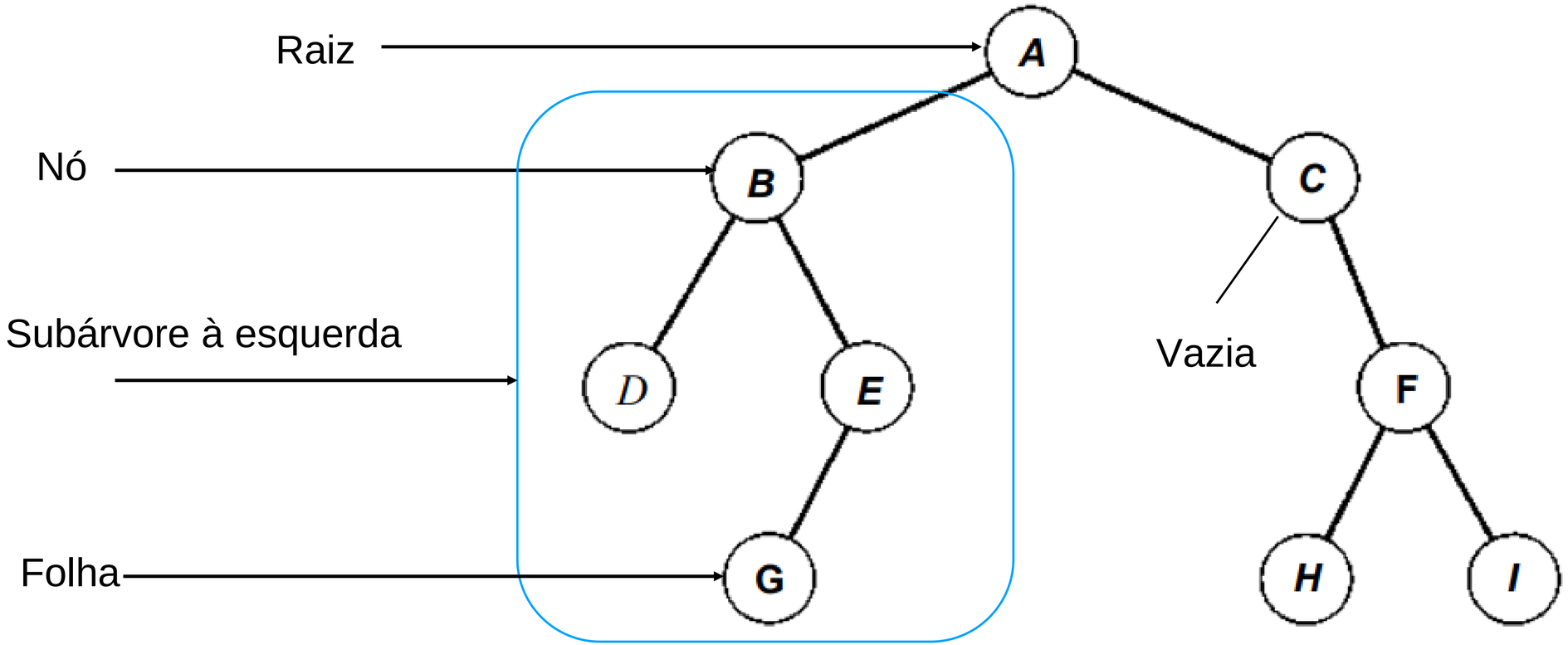


Figura 5.1.1 Uma árvore binaria.

Árvores

- Conjunto finito e não-vazio de elementos, no qual um elemento é chamado raiz e os elementos restantes são particionados em m >= 0 subconjuntos disjuntos, cada um dos quais sendo uma árvore em si mesmo.

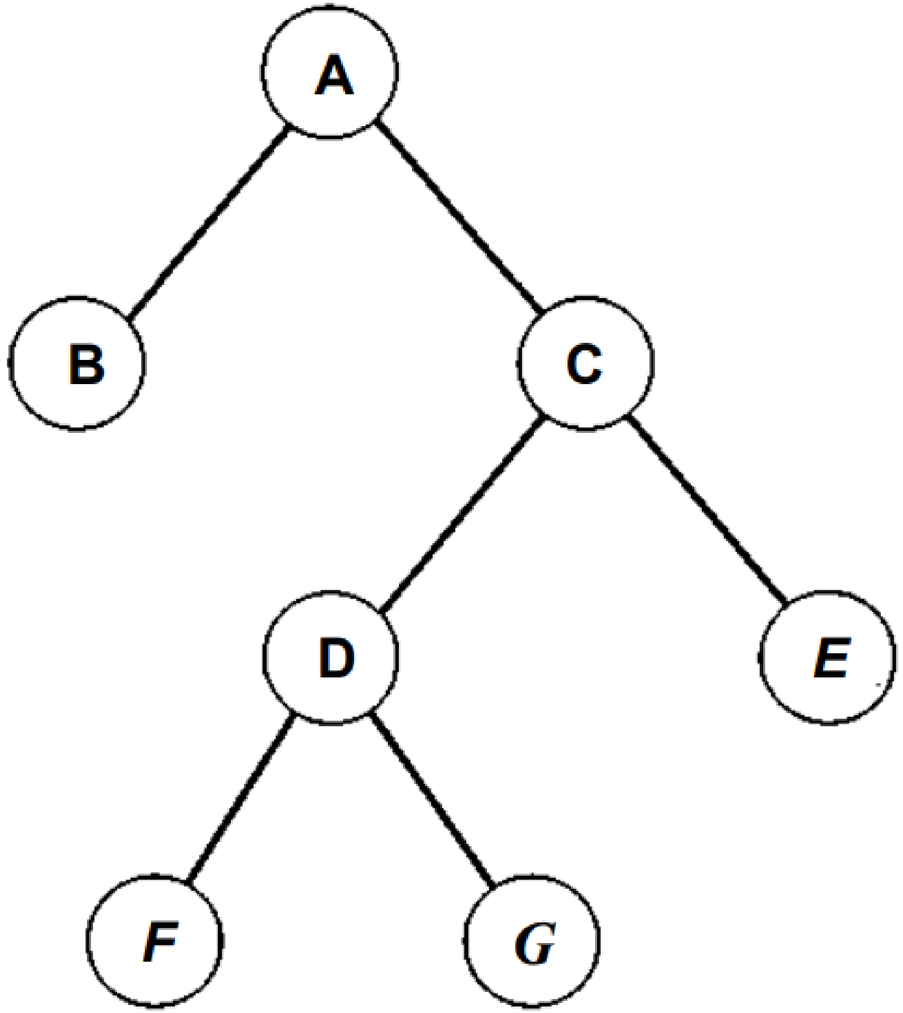
- Cada elemento numa árvore é chamado nó da árvore.

- Cada nó pode ser a raiz de uma árvore com zero ou mais subárvores. Um nó sem subárvores é uma folha. Usamos os termos/mi, filho, irmão, ancestral, descendente, nível e profundidade com a mesma conotação utilizada para as árvores binárias

Árvores

- Se todo nó que não é folha numa árvore binária tiver subárvores esquerda e direita não-vazias, a árvore será considerada uma **árvore estritamente binária**

- O nível de um nó numa árvore binária é definido como segue: **a raiz da árvore tem nível 0**, e o nível de qualquer **outro nó** na árvore é **um nível a mais que o nível de seu pai**.

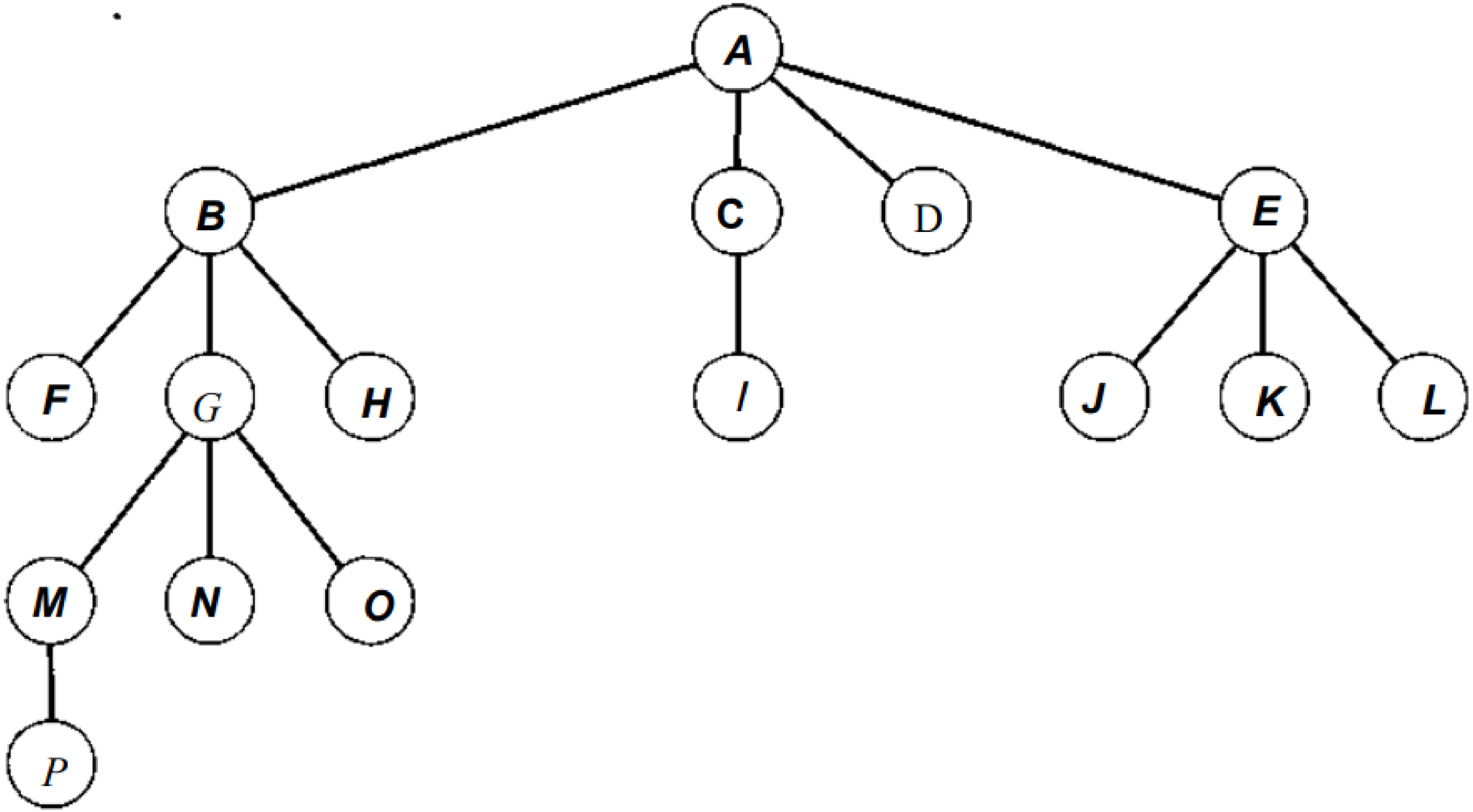


Árvores

- A **profundidade** de uma árvore binária significa o nível máximo de qualquer folha na árvore.

- Uma **árvore binária completa de profundidade *d*** é a árvore estritamente binária em que todas as folhas estejam no nível *d*.

Árvores



Análise descendente

- A análise descendente é realizada a partir da derivação mais à esquerda
- Realiza-se a partir da raiz para as folhas da árvore – em abordagem *top-down*
- Pode ser realizada de duas formas: **análise de *tracking* à retaguarda** ou **preditiva**
- A análise preditiva tenta predizer a próxima construção usando *tokens lookahead*

Análise descendente

- A análise de *tracking* à retaguarda tenta diferentes possibilidades para o *token* de entrada

- Estudaremos dois tipos de algoritmos de análise:

  - **análise descendente recursiva** e

  - **análise LL(1)**, (L, processa entradas da esquerda para a direita; L, traça uma derivação mais à esquerda para a cadeia de entrada; 1 , usando somente 1 símbolo de entrada para determinar a direção da árvore.

- Há vantagens e desvantagens em cada abordagem

Método recursivo descendente

- Utiliza a BNF ou EBNF para orientar a construção do código
- **Toma o lado direito da sentença da gramática** para implementar a estrutura do código
- A **sequência de terminais e não terminais** em uma escolha corresponde à combinação de entradas e chamadas para procedimentos (*procedures*), enquanto as escolhas correspondem às alternativas

Método recursivo descendente

- **Exemplo 1**: seja a gramática

  exp → exp soma termo | termo

  soma → + | -

  termo → termo mult fator | fator

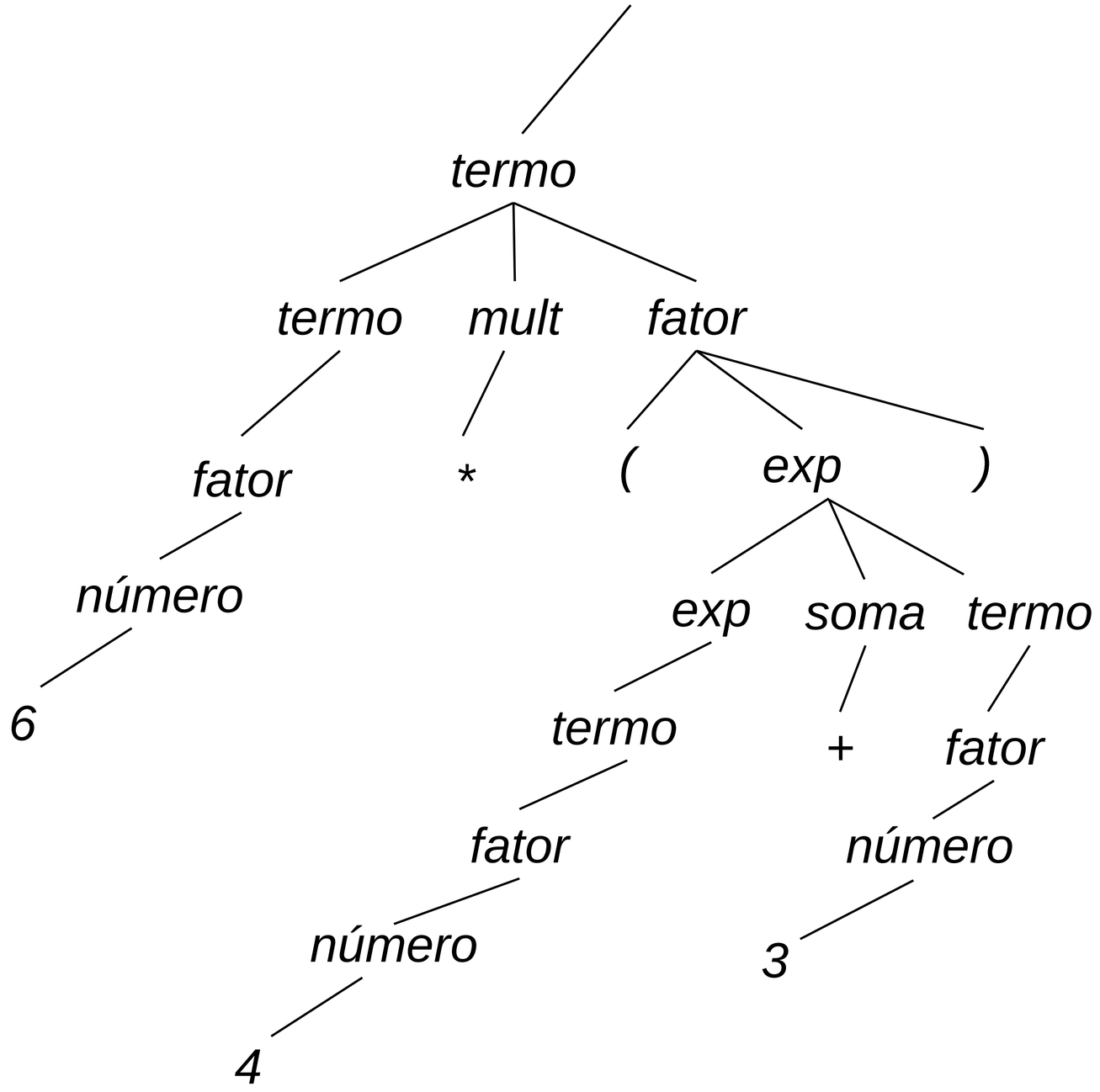
  *mult → **

  fator → (exp) | número

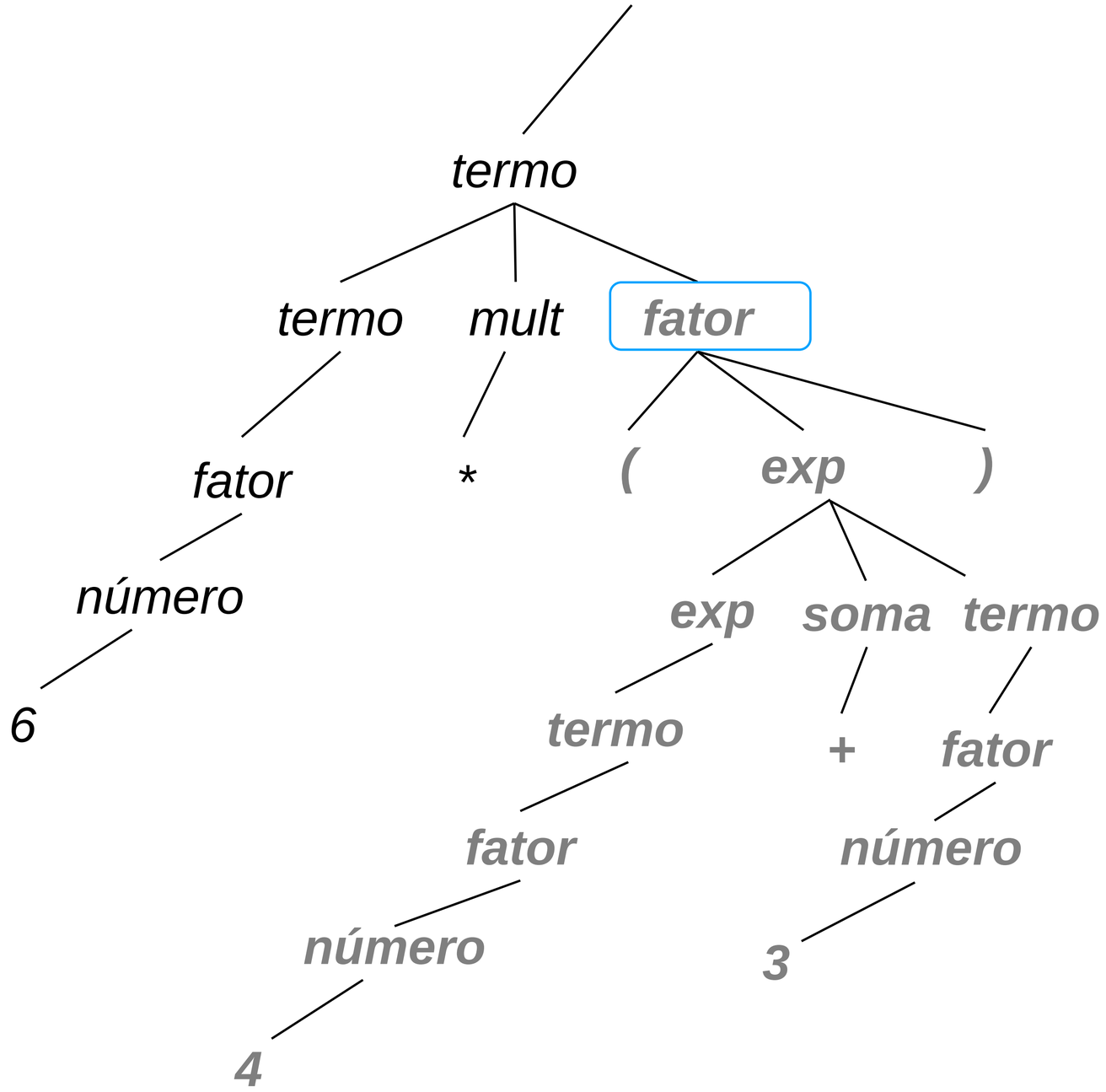
  número → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

E a sentença *6 * (4 + 3)*

Dada a sentença: *6 * (4 + 3)*



Dada a sentença: 6 * (4 + 3)



Método recursivo descendente

- Um **exemplo** de procedimento recursivo descendente, no arquivo ***analiseFator***, em C
  - Existe uma variável *token* que registra o **próximo** *token* de entrada;
  - Usa-se apenas **um símbolo de verificação à frente...**;
  - Existe procedimento (*procedure*) ***casamento()*** que casa o próximo *token* com seu argumento...;
  - e **avança a entrada** ao próximo *token* se a sentença está correta ou ...;
  - declara erro em caso contrário.

Método recursivo descendente

- Observe que a *procedure* ***erro()*** não possui especificação, produz somente a mensagem de erro...;
- após a mensagem de erro a *procedure* prosseguiu chamando o próximo *token* e continuando a análise...;
- e que a procedure ***exp()*** ativará as procedures ***termo(), fator() e soma() – conforme a árvore de análise sintática***

Método recursivo descendente

- **Exemplo 2**: seja a gramática

if-decl → *if (exp) declaração*

| if (exp) declaração else declaração

exp → *idt oplog idt*

idt → *[a – z]*

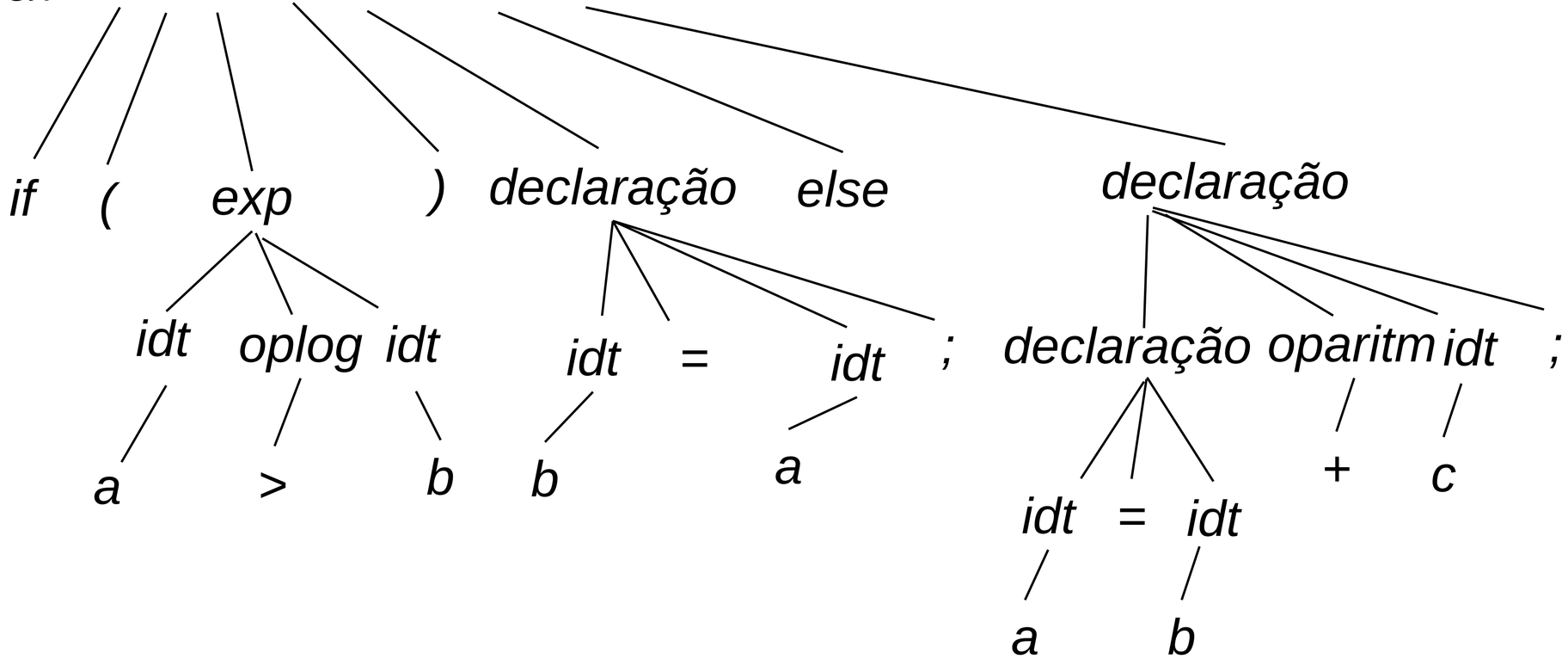
oplog → *> | < | == | !=*

declaração → *declaração oparitm idt | declaração; | idt = idt*

oparitm → *+ | - | * | /*

E a sentença *if (a>b) b=a; else a=b + c;*

Dada a sentença: *if (a>b) b=a; else a=b + c;*



Método recursivo descendente

- Um **exemplo** de procedimento recursivo descendente, no arquivo ***analiseIF***, em C
  - As *procedures* obedecem à gramática,...;
  - que possui ambiguidades, como na construção do *else*...;
  - assim como na abstração da *declaração*

Método LL(1)

- Utiliza uma pilha explícita, ao invés de ativações recursivas
- Representa-se a pilha de uma forma padrão
- As ações podem ser mostradas em uma tabela com quatro colunas:
  1. Enumeração dos passos para referência posterior
  2. **Pilha de análise**, que mostra o conteúdo da pilha – **a regra de gramática**, com o final da pilha **($)** à esquerda e o topo à direita;
  3. **Entrada**, que mostra os símbolos de entrada – **sentença a analisar**, apresentados da esquerda para a direita – o **$ indica o final**
  4. **Ação**, que apresenta uma descrição da ação do analisador - **código**

Método LL(1)

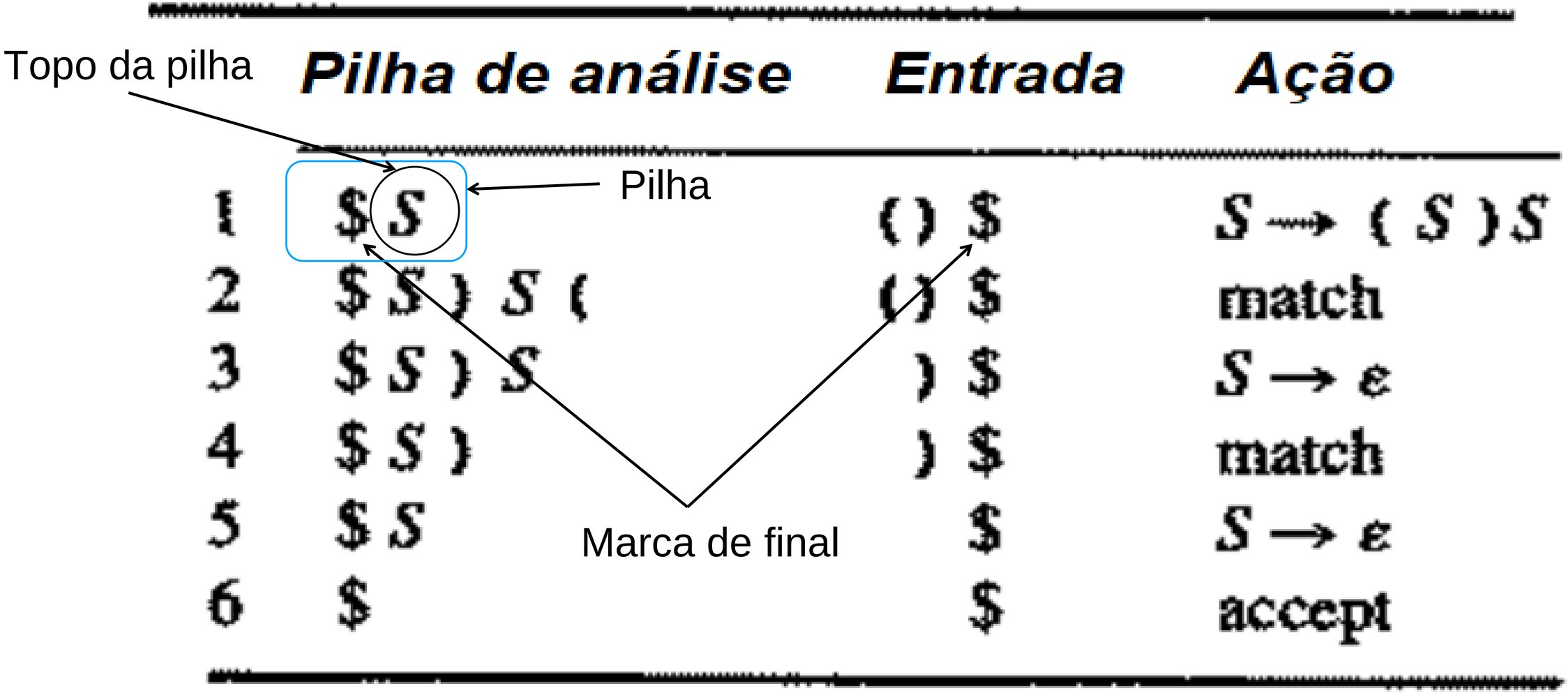
- Seja o exemplo da regra de gramática

$$S \rightarrow (S) S \mid \varepsilon$$

, e a cadeia de caracteres ( ).

- A tabela é dada por...

Método LL(1)

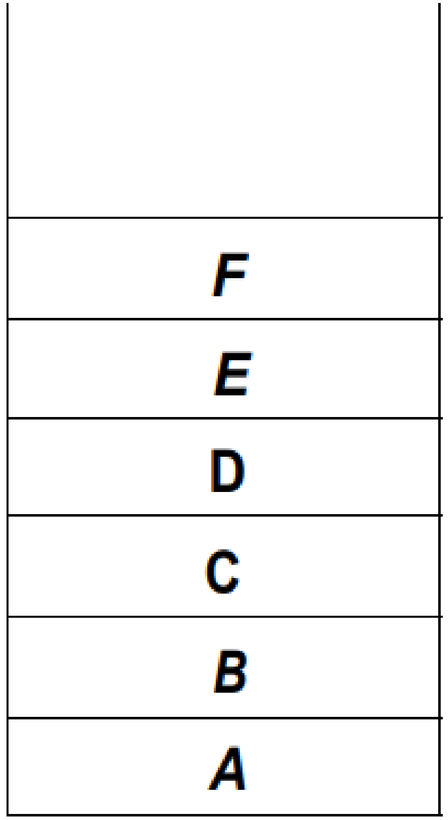


	Pilha de análise	Entrada	Ação
1	$ S	( ) $	S → (S) S
2	$ S) S (	( ) $	match
3	$ S) S	) $	S → ε
4	$ S)	) $	match
5	$ S	$	S → ε
6	$	$	accept

Pilha

- Uma pilha é um **conjunto ordenado** de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados itens em uma extremidade chamada **topo** da pilha. Objeto **dinâmico** e **mutável**.

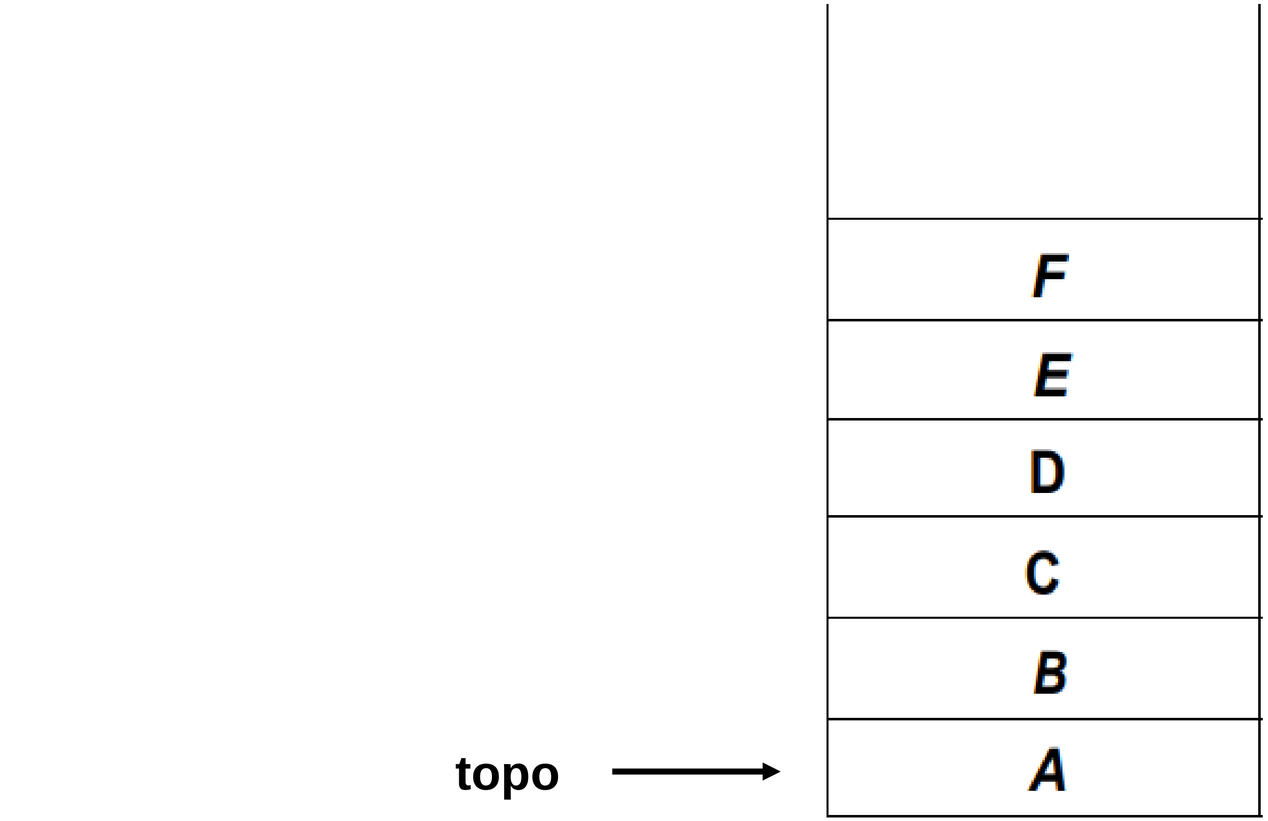
- Operações primitivas:
  - *push()*
  - *pop()*



Uma pilha contendo termos da pilha.

Pilha

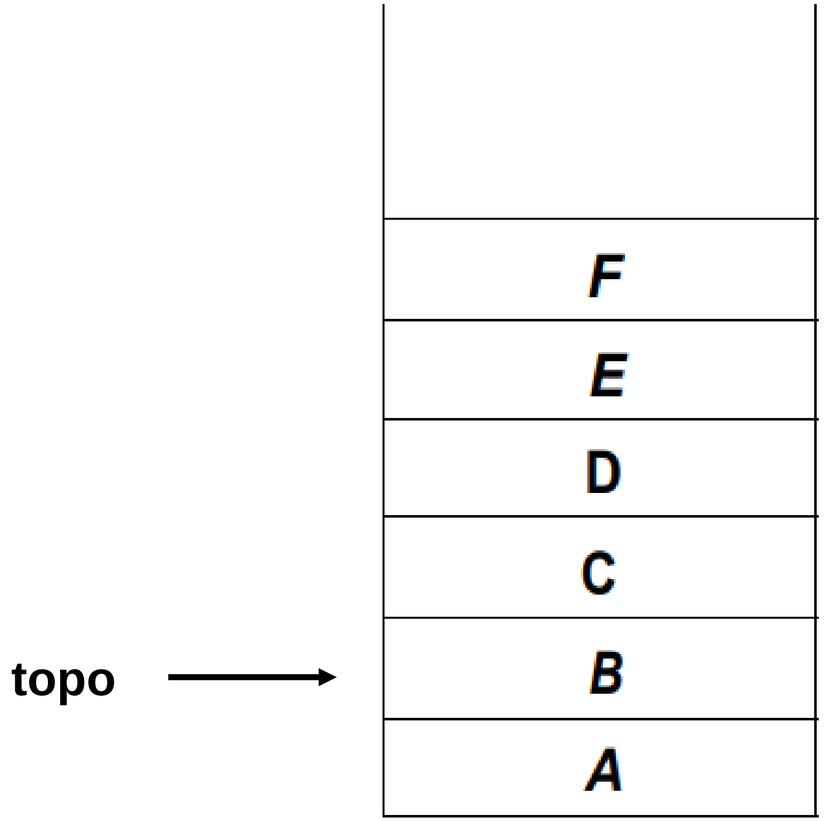
- *push(A)*
- *push(B)*
- *push(C)*
- *push(D)*
- *push(E)*
- *push(F)*



Uma pilha contendo termos da pilha.

Pilha

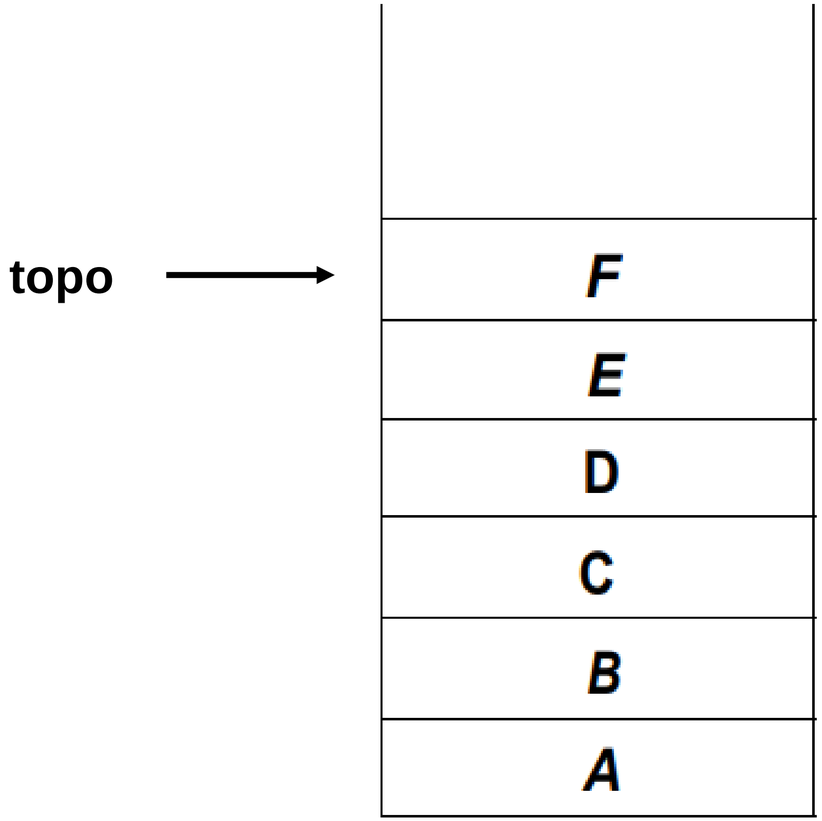
- *push(A)*
- *push(B)*



Uma pilha contendo termos da pilha.

Pilha

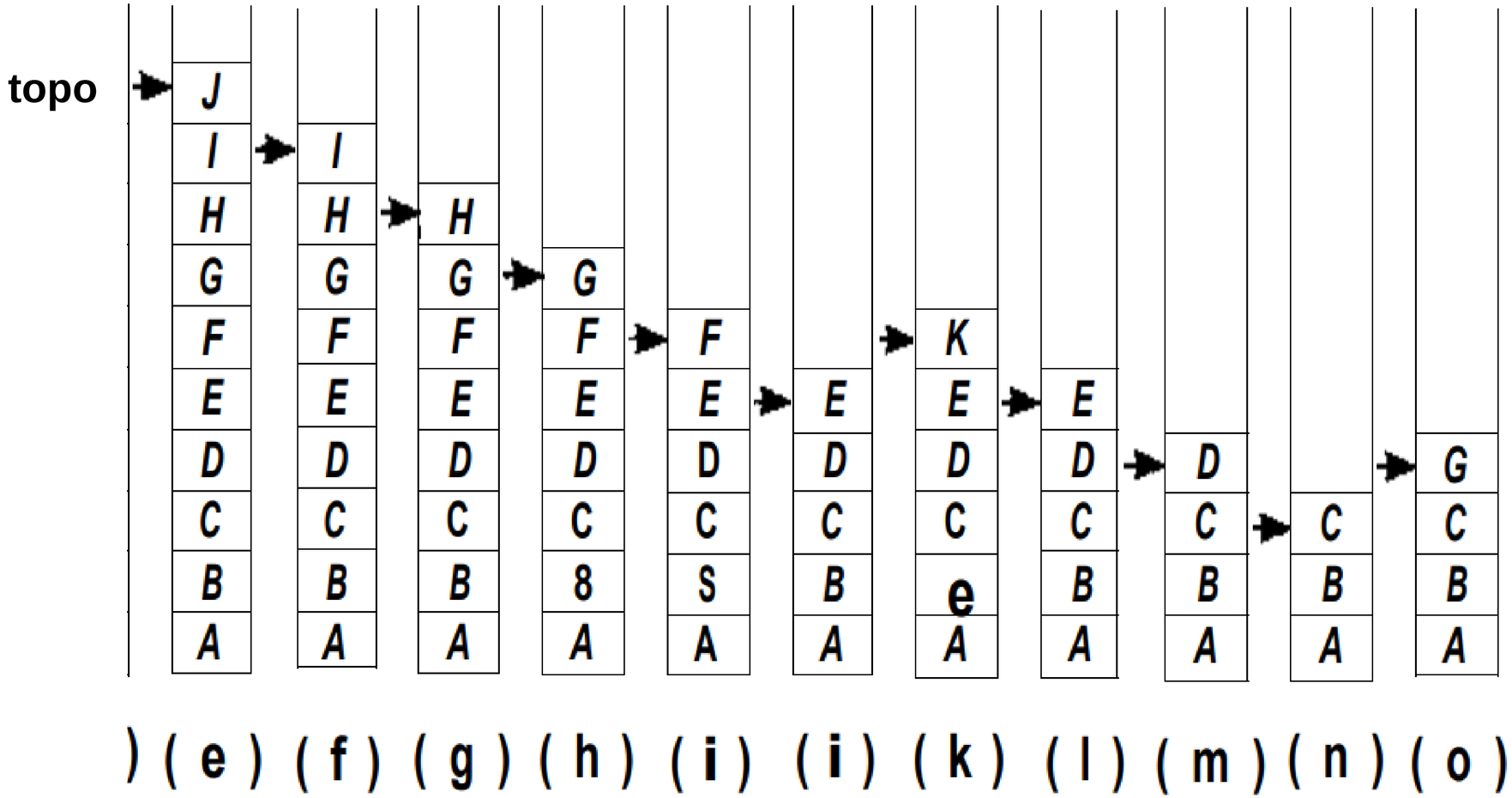
- *push(A)*
- *push(B)*
- *push(C)*
- *push(D)*
- *push(E)*
- *push(F)*



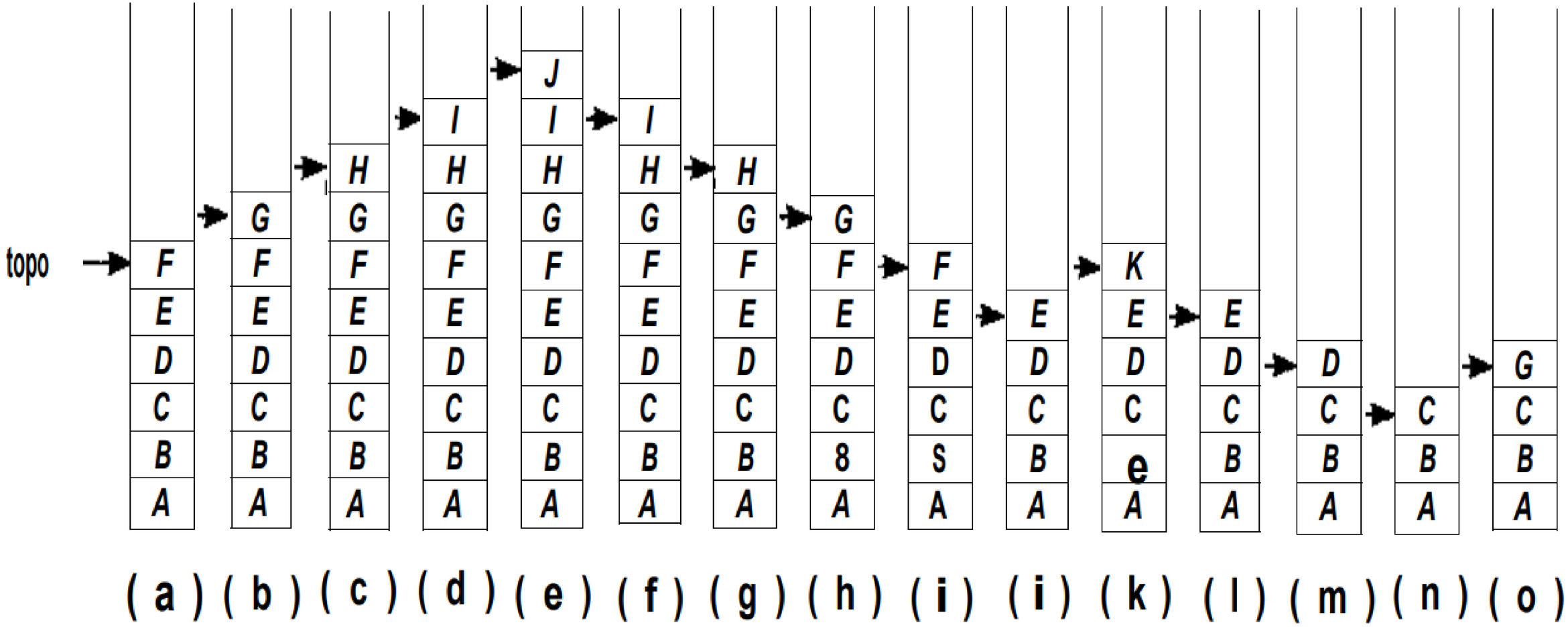
Uma pilha contendo termos da pilha.

Pilha

- *push(A)*
- *push(B)*
- *push(C)*
- *push(D)*
- *push(E)*
- *push(F)*
- ...
- *push(I)*
- *pop()*
- *pop()*
- ...



Pilha


topo
F E D C B A
G F E D C B A
H G F E D C B A
I H G F E D C B A
J I H G F E D C B A
I H G F E D C B A
H G F E D C B A
G F E D C B 8 A
F E D C S A
E D C B A
K E D C e A
E D C B A
D C B A
C B A
G C B A
(a) (b) (c) (d) (e) (f) (g) (h) (i) (i) (k) (l) (m) (n) (o)

Pilha

- Estrutura

```
typedef struct
int topo;
char *items[PILHA_TAM];
} pilha;
pilha *pilhaArvore;
```

LL(1): passo a passo

1. O analisador começa pela colocação do símbolo de início na pilha...

2. Aceita uma cadeia de entrada se, após uma série de ações, a pilha e a entrada forem vazias...

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	$S \rightarrow (S) S$

...

3. **Gera** um não terminal no **topo da pilha** por uma de suas escolhas da regra gramatical...

Método LL(1)

- A ação gera o *token* corrente no topo da pilha, caso tenha casado com o token de entrada, descartando tanto da pilha como da entrada

3. **Gera** um não terminal por uma cadeia da regra **ou**

4. **Casa** o *token* no top da pilha com o próximo

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	S → (S) S
$ S) S (	( ) $	casa (da entrada com)
…		

LL(1): passo a passo

5. Na construção da árvore de análise sintática, são inseridos nós à medida que a árvore é construída.

Método LL(1)

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	$S \rightarrow (S) S$
$ S) S (	( ) $	*casa (da entrada com)*
$ S) S	) $	ε) S
...		

Método LL(1)

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	$S \rightarrow (S) S$
$ S) S (	( ) $	*casa (da entrada com)*
$ S) S	) $	ε) S
$ S) S	ε) $	*casa* ε *da entrada com S*
…		

Método LL(1)

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	$S \rightarrow (S) S$
$ S) S (	( ) $	*casa (da entrada com)*
$ S) S	) $	ε) S
$ S) S	ε) $	*casa* ε *da entrada com S*
$ S)	) $	) S
…		

Método LL(1)

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	$S \rightarrow (S) S$
$ S) S (	( ) $	*casa (da entrada com (*
$ S) S	) $	ε) S
$ S) S	ε) $	*casa* ε *da entrada com S*
$ S)	) $	) S
$ S)	) $	*casa) da entrada com)*
...		

Método LL(1)

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	$S \rightarrow (S) S$
$ S) S (	( ) $	*casa (da entrada com (*
$ S) S	) $	ε) S
$ S) S	ε) $	*casa* ε *da entrada com S*
$ S)	) $	) S
$ S)	) $	*casa) da entrada com)*
$ S	$	ε
...		

Método LL(1)

No exemplo,

Pilha de análise	*Entrada*	*Ação*
$ S	( ) $	$S \rightarrow (S) S$
$ S) S (	( ) $	*casa (da entrada com (*
$ S) S	) $	ε) S
$ S) S	ε) $	*casa* ε *da entrada com S*
$ S)	) $	) S
$ S)	) $	*casa) da entrada com)*
$ S	$	ε
$ S	ε $	*casa* ε *da entrada com S*

ACEITAÇÃO

Tabela de análise sintática LL(1)

M[N,T]

- Matriz bidimensional indexada por **não terminais** e **terminais** com escolhas, onde *N* é o conjunto de não terminais e *T* é conjunto de terminais ou *tokens*
- A tabela inicialmente é uma **tabela vazia**
- As células que permanecem vazias após a sua construção representam erros potenciais que podem ocorrer durante a análise sintática

Tabela de análise sintática LL(1)

M[N,T]

- Construção da tabela de acordo com as regras
  1. Dado um *token* ***a*** seleciona-se uma regra de gramática $A \rightarrow \beta$ para obter um casamento
  2. Se a regra de gramática *A* derivar a cadeia vazia e houver um *token* ***a*** que possa suceder a derivação

Considerando o exemplo, $S \rightarrow (\;) \; \$$

M[N,T]	(	)	$
S	$S \rightarrow (\; S \;) \; S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Método LL-1

- **Exemplo**: seja a gramática

  if-decl → *if (exp) declaração*

  | *if (exp) declaração else declaração*

  exp → *idt oplog idt*

  idt → *[a – z]*

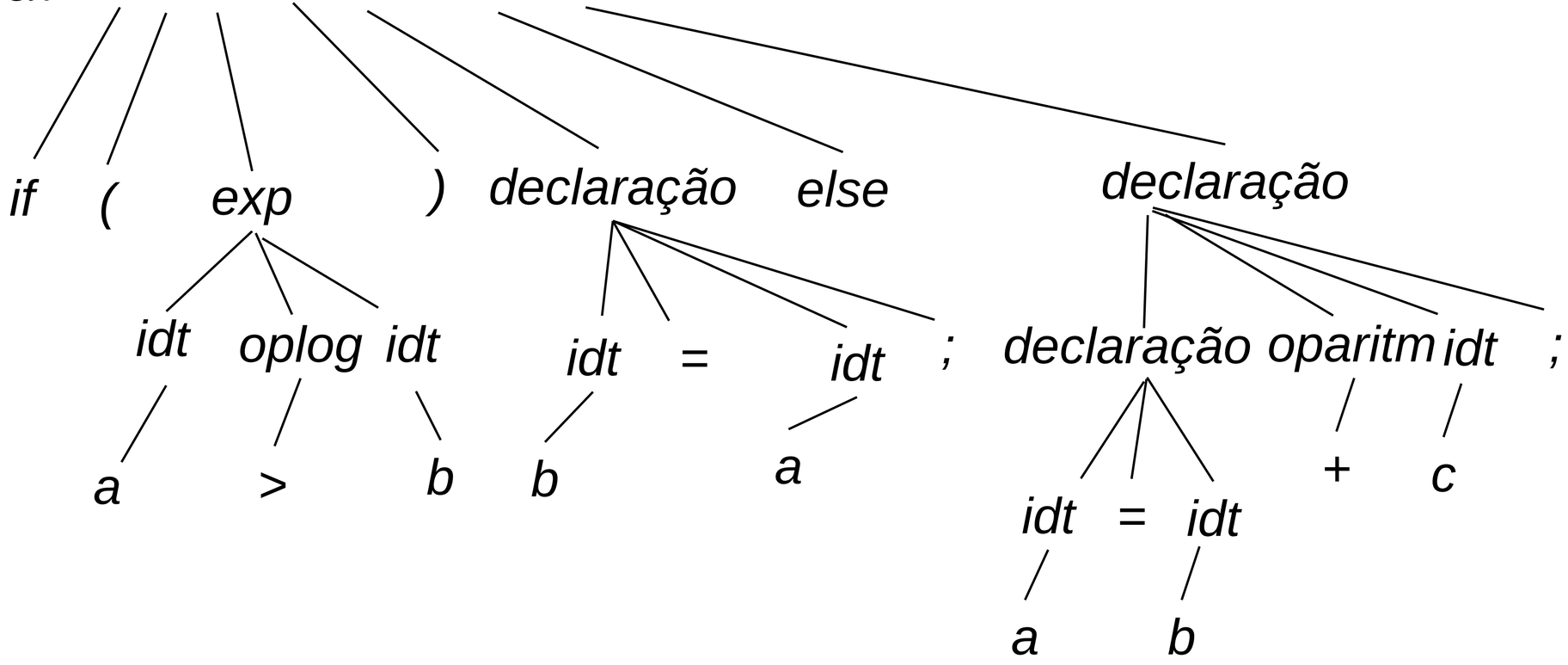
  oplog → *> | < | == | !=*

  declaração → *declaração oparitm idt | declaração; | idt = idt*

  oparitm → *+ | - | * | /*

E a sentença *if (a>b) b=a; else a=b + c;*

Dada a sentença: *if (a>b) b=a; else a=b + c;*



Método LL-1

- Um **exemplo** de procedimento recursivo descendente, no arquivo ***analiseIF2***, em C
  - A pilha é criada no início de acordo com a abstração da qual se obterá a derivação,...;
  - A derivação está feita à esquerda, ou seja, decompondo-se as abstrações da esquerda para a direita.
  - Busca-se o casamento da cadeia de entrada com os terminais da pilha de abstração...;
  - Se houver casamento busca-se o próximo *token*...

Método LL-1

- Um **exemplo** de procedimento recursivo descendente, no arquivo ***analiseIF2***, em C
  - ...e realiza-se uma derivação até obter um novo símbolo terminal na pilha.
  - Se não houver casamento chama-se a procedure erro() e busca-se o próximo *token*.
  - Repete-se o processo até alcançar o símbolo $ na pilha e na cadeia de entradas.

IBMEC.BR

/IBMEC

IBMEC

@IBMEC_OFICIAL

@IBMEC

Ibmec