Conteúdo Programático e Cronograma

1º Bimestre:

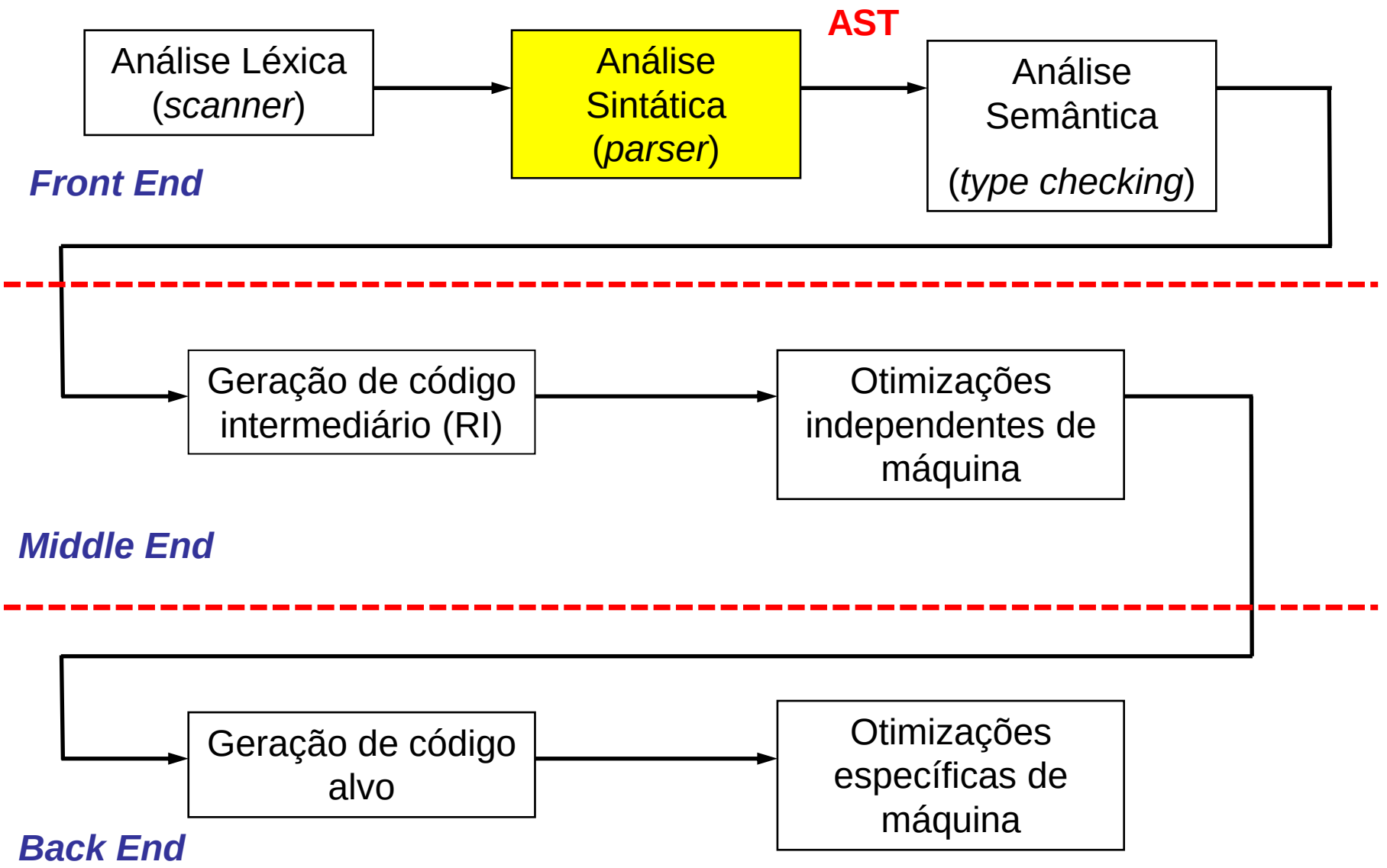
- ~~Organização e estrutura de compiladores~~
- ~~Análise Léxica~~
- Análise Sintática
- Ferramentas de geração automática de compiladores

2º Bimestre:

- Análise Semântica

Análise Sintática

Fluxo do Compilador



Analisador Sintático (*Parser*)

- Recebe uma seqüência de tokens do analisador léxico e determina se a *string* pode ser gerada através da gramática da linguagem fonte.

- É esperado que ele reporte os erros de uma maneira inteligível.

- Deve se recuperar de erros comuns, continuando a processar a entrada.

Analisador Sintático (*Parser*)

- **ER**s são boas para definir a estrutura léxica de maneira declarativa.

- Será que são “poderosas” o suficiente para conseguir definir declarativamente a estrutura sintática de linguagens de programação ???

Analisador Sintático (*Parser*)

- As **ER**s devem ser capazes de expressar a sintaxe de linguagens de programação.

- E se forem dados nomes para abreviar as **ER**s?

EXPR = ab(c|d)e

EXPR = a b **AUX** e
AUX = c | d

Analisador Sintático (*Parser*)

- Exemplo de **ER** usando abreviações:

  digits = $[0\text{-}9]^+$

  sum = (digits “+”)* digits

  definem somas da forma **28+301+9**

- Como isso é implementado?
  - O analisador léxico substitui as abreviações antes de traduzir para um autômato finito
  - sum = ($[0\text{-}9]^+$ “+”)* $[0\text{-}9]^+$

Analisador Sintático (*Parser*)

- É possível usar a mesma idéia para definir uma linguagem para expressões que tenham parênteses balanceados?

  (1+(245+2))

- Tentativa:

  digits = [0-9]$^{+}$

  sum = expr “+” expr

  expr = “(” sum “)” | digits

Analisador Sintático (*Parser*)

digits = [0-9]$^{+}$
sum = expr “+” expr
expr = “(” sum “)” | digits

- O analisador léxico substituiria *sum* em *expr:*

  expr = “(” expr “+” expr “)” | digits

- Depois substituiria expr no próprio expr:

  expr = “(” **(** “(” expr “+” expr “)” | digits **)** “+” expr “)” | digits

- Continua tendo *expr’s* do lado direito!

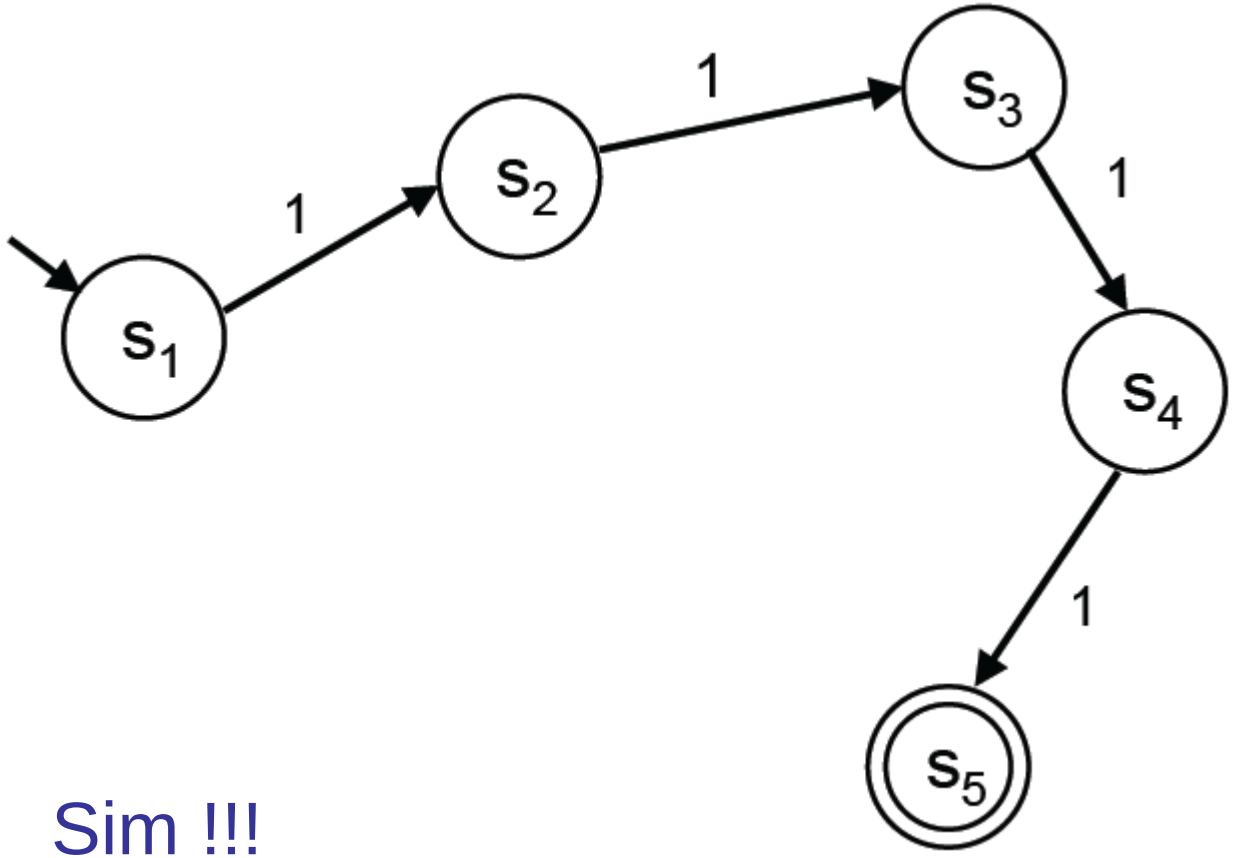
Analisador Sintático (*Parser*)

- As abreviações não acrescentam a ERs o poder de expressar recursão.

- É isso que se precisa para expressar a recursão mútua entre sum e expr e também expressar a sintaxe de linguagens de programação.

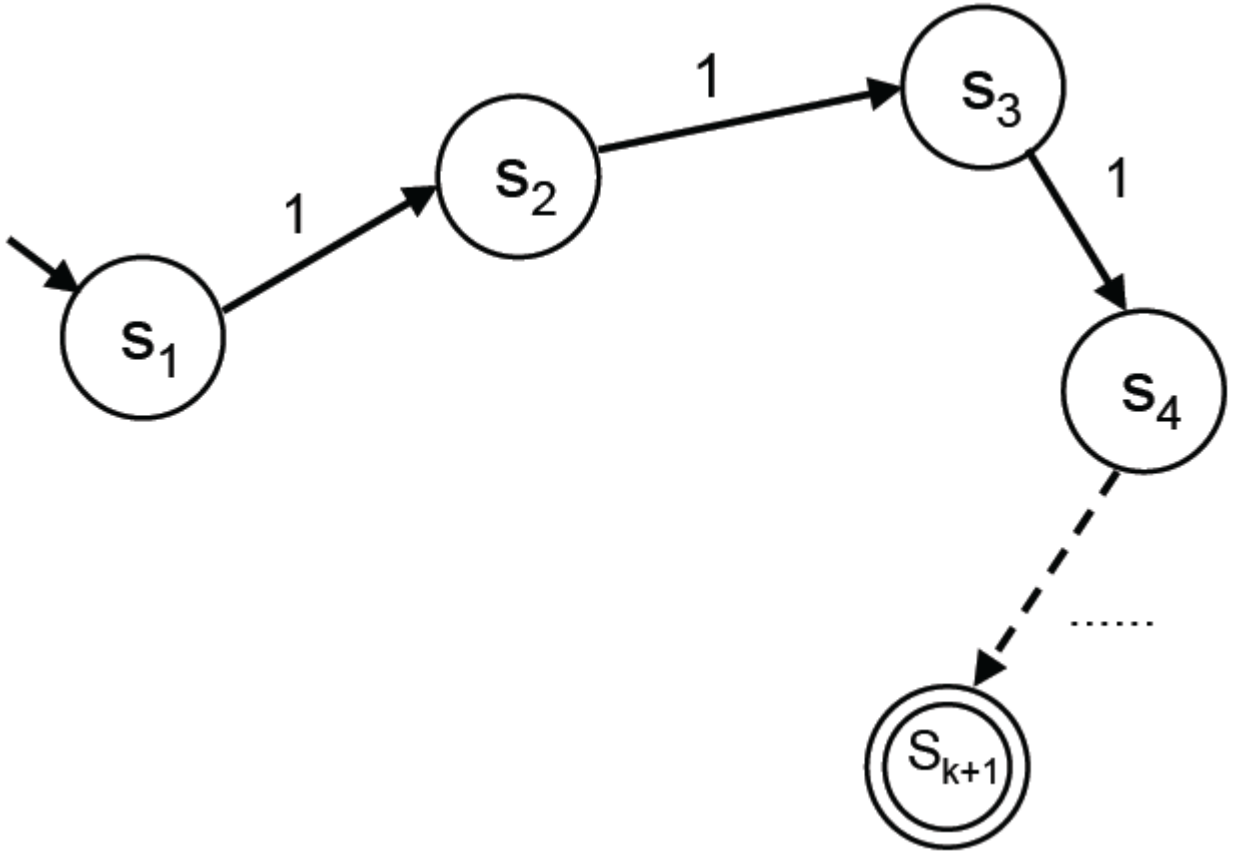
- **O que está faltando?**

É possível contar 4 “1”s com um DFA?

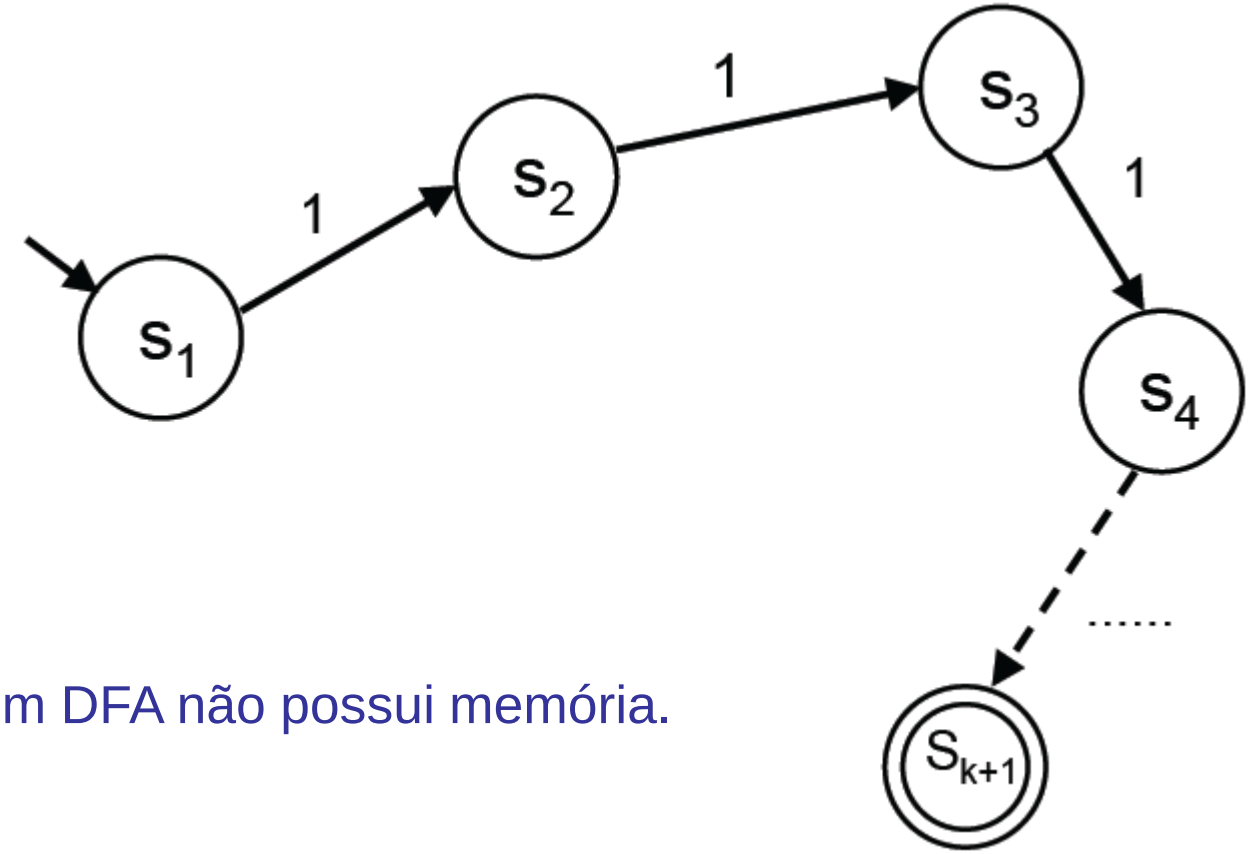


Sim !!!

É possível contar k “1”s com um DFA?

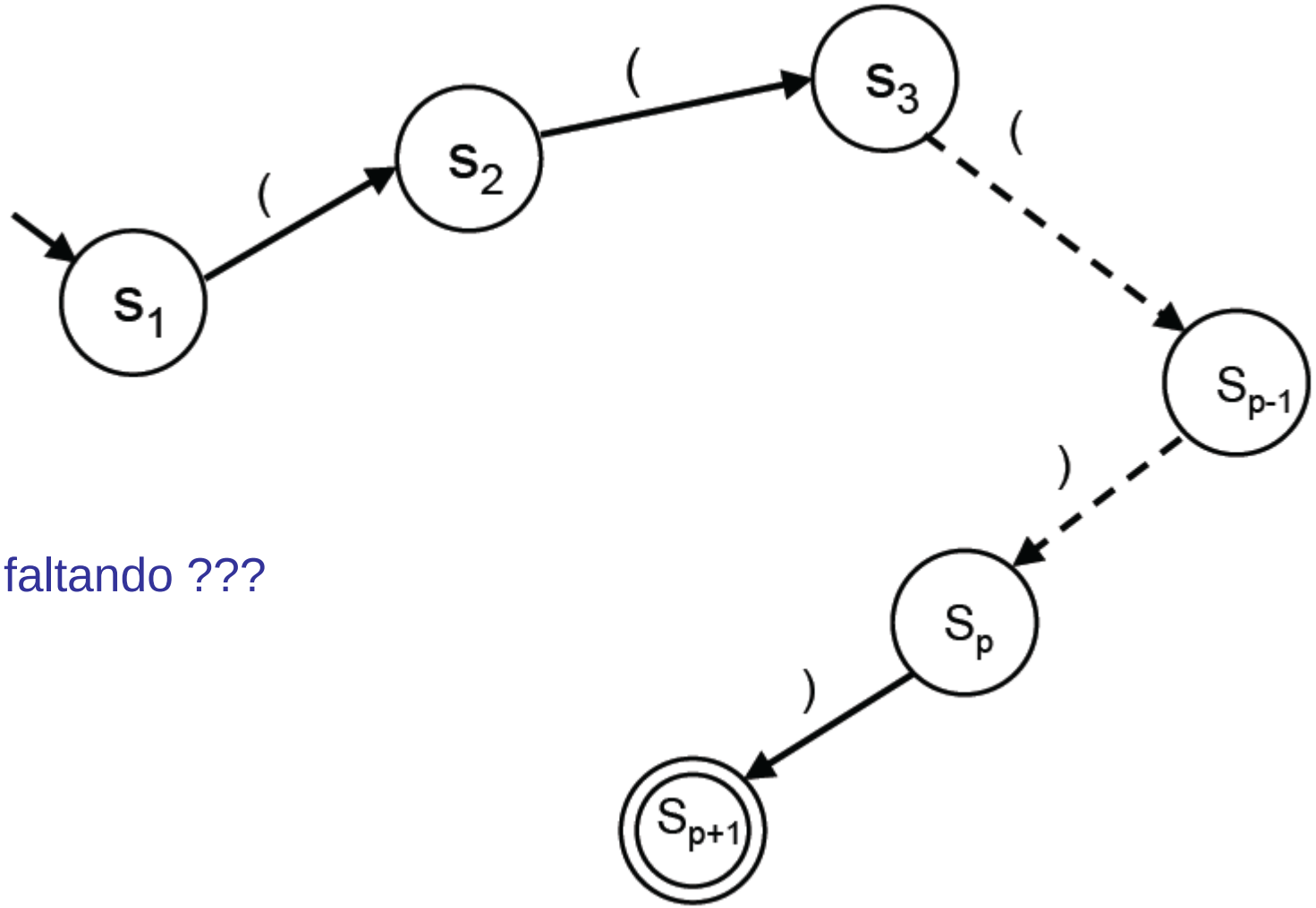


É possível contar k “1”s com um DFA?



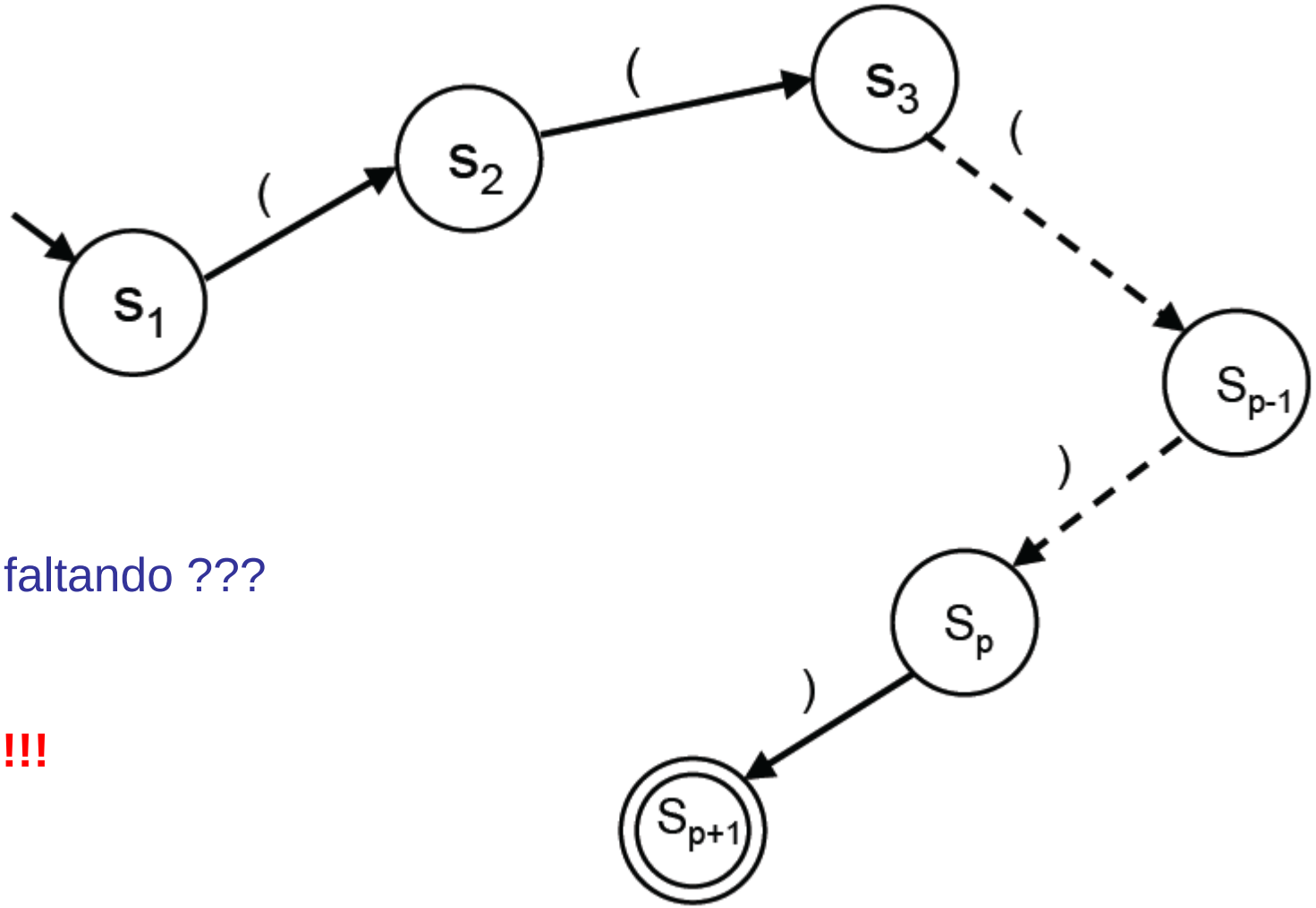
Não, pois um DFA não possui memória.

Como então casar ((…)) para um k qualquer?



O que está faltando ???

Como então casar ((…)) para um k qualquer?

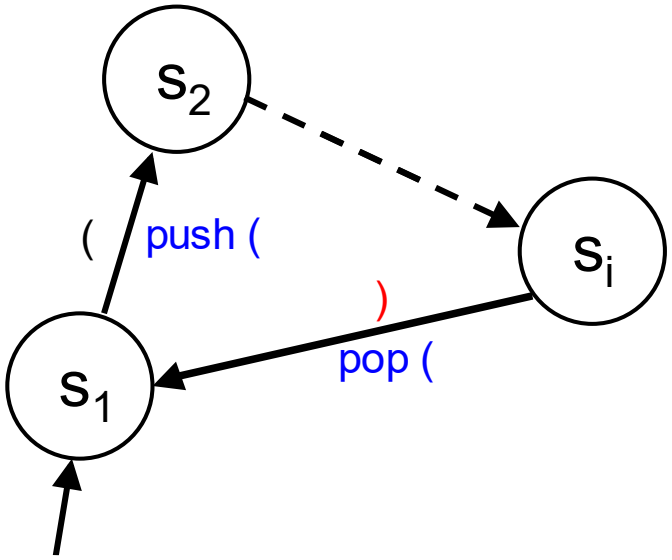


O que está faltando ???

Uma Pilha !!!

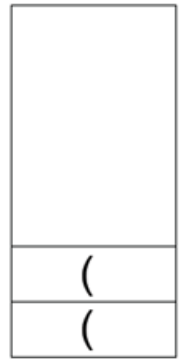
Contando com uma Pilha

input: ((.....)) $



token: (
ação: push (

token: (
ação: push (



. . .

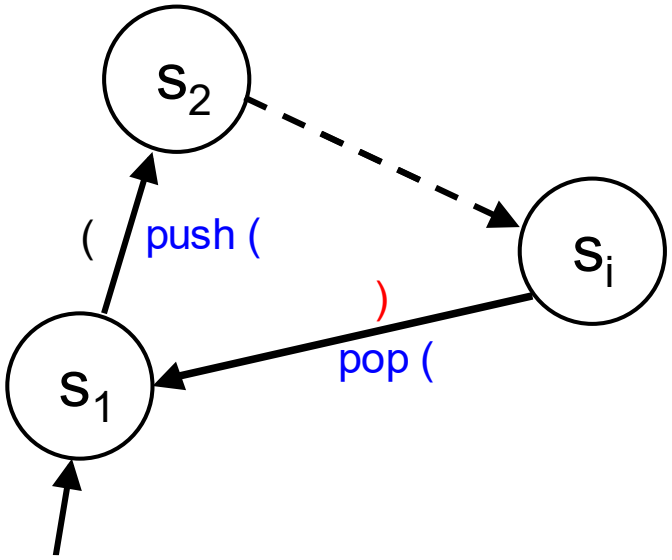
token:)
ação: pop (

token:)
ação: pop (

termina
vazia

Contando com uma Pilha

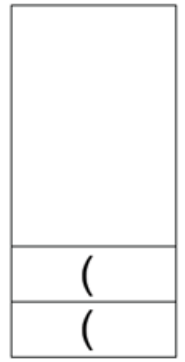
input: ((.....)) $



Uma autômato com pilha corresponde a que?

token: (
ação: push (

token: (
ação: push (



. . .

token:)
ação: pop (

token:)
ação: pop (

termina
vazia

Context-Free Grammar (Gramática Livre de Contexto)

Descrevem uma linguagem através de um conjunto de produções da forma:

symbol -> symbol symbol symbol … symbol

onde existem zero ou mais símbolos no lado direito.

Produções funcionam como regras de substituição:

Símbolos:

- **terminais**: pertencem ao alfabeto da linguagem
- **não-terminais**: aparecem do lado esquerdo de alguma produção
- Nenhum **terminal** aparece do lado esquerdo de uma produção
- Existe um **não-terminal** definido como **símbolo inicial.** Normalmente é o da primeira regra

Context-Free Grammar

- Gerar cadeias da linguagem:
  1. Escreva a variável inicial.
  2. Encontre uma variável escrita e uma regra para essa variável. Substitua essa variável pelo lado direito da regra.
  3. Repita 2 até não restar variáveis

$$
\begin{aligned}
&1.\ A \rightarrow 0A1 \\
&2.\ A \rightarrow B \\
&3.\ B \rightarrow \#
\end{aligned}
$$

Context-Free Grammar

A sequência de substituições é chamada de derivação.

Ex:

$A \rightarrow 0A1$

$A \rightarrow B$

$B \rightarrow \#$

- 000#111
- A → 0A1 → 00A11 → 000A111→ 000B111→ 000#111

Linguagem: O conjunto de todas as cadeias que podem ser geradas dessa maneira

Context-Free Grammar

1. *SENTENCE* → *NOUN-PHRASE VERB-PHRASE*
2. *NOUN-PHRASE* → *CMPLX-NOUN* | *CMPLX-NOUN PREP-PHRASE*
3. *VERB-PHRASE* → *CMPLX-VERB* | *CMPLX-VERB PREP-PHRASE*
4. *PREP-PHRASE* → *PREP CMPLX-NOUN*
5. *CMPLX-NOUN* → *ARTICLE NOUN*
6. *CMPLX-VERB* → *VERB* | *VERB NOUN-PHRASE*
7. *ARTICLE* → a | the
8. *NOUN* → boy | girl | flower
9. *VERB* → touches | likes | sees
10. *PREP* → with

Como é a derivação para:

a boy sees

Context-Free Grammar

1. $S \rightarrow S$ `;` S
2. $S \rightarrow$ `id :=` E
3. $S \rightarrow$ `print(`L`)`
4. $E \rightarrow$ `id`
5. $E \rightarrow$ `num`
6. $E \rightarrow E$ `+` E
7. $E \rightarrow$ `(`S`,` E`)`
8. $L \rightarrow E$
9. $L \rightarrow L$ `,` E

```
id := num; id := id + (id := num + num, id)
```

Possível código fonte:

```
a : = 7; b : = c + (d : = 5 + 6, d)
```

Derivações

```
a : = 7; b : = c + (d : = 5 + 6, d)
```

S
S ; *S*
S ; id := *E*
id := *E*; id := *E*
id := num ; id := *E*
id := num ; id := *E* + *E*
id := num ; id := *E* + (*S*, *E*)
id := num ; id := id + (*S*, *E*)
id := num ; id := id + (id := *E*, *E*)
id := num ; id := id + (id := *E* + *E*, *E*)
id := num ; id := id + (id := *E* + *E*, id)
id := num ; id := id + (id := num + *E*, id)
id := num ; id := id + (id := num + num, id)

1. *S* → *S* **;** *S*
2. *S* → **id :=** *E*
3. *S* → **print(***L***)**
4. *E* → **id**
5. *E* → **num**
6. *E* → *E* **+** *E*
7. *E* → **(***S*, *E***)**
8. *L* → *E*
9. *L* → *L* , *E*

Derivações

- ***left-most****:* o não terminal mais a esquerda é sempre o expandido;

- ***right-most****:* idem para o mais a direita.

- Qual é o caso do exemplo anterior?

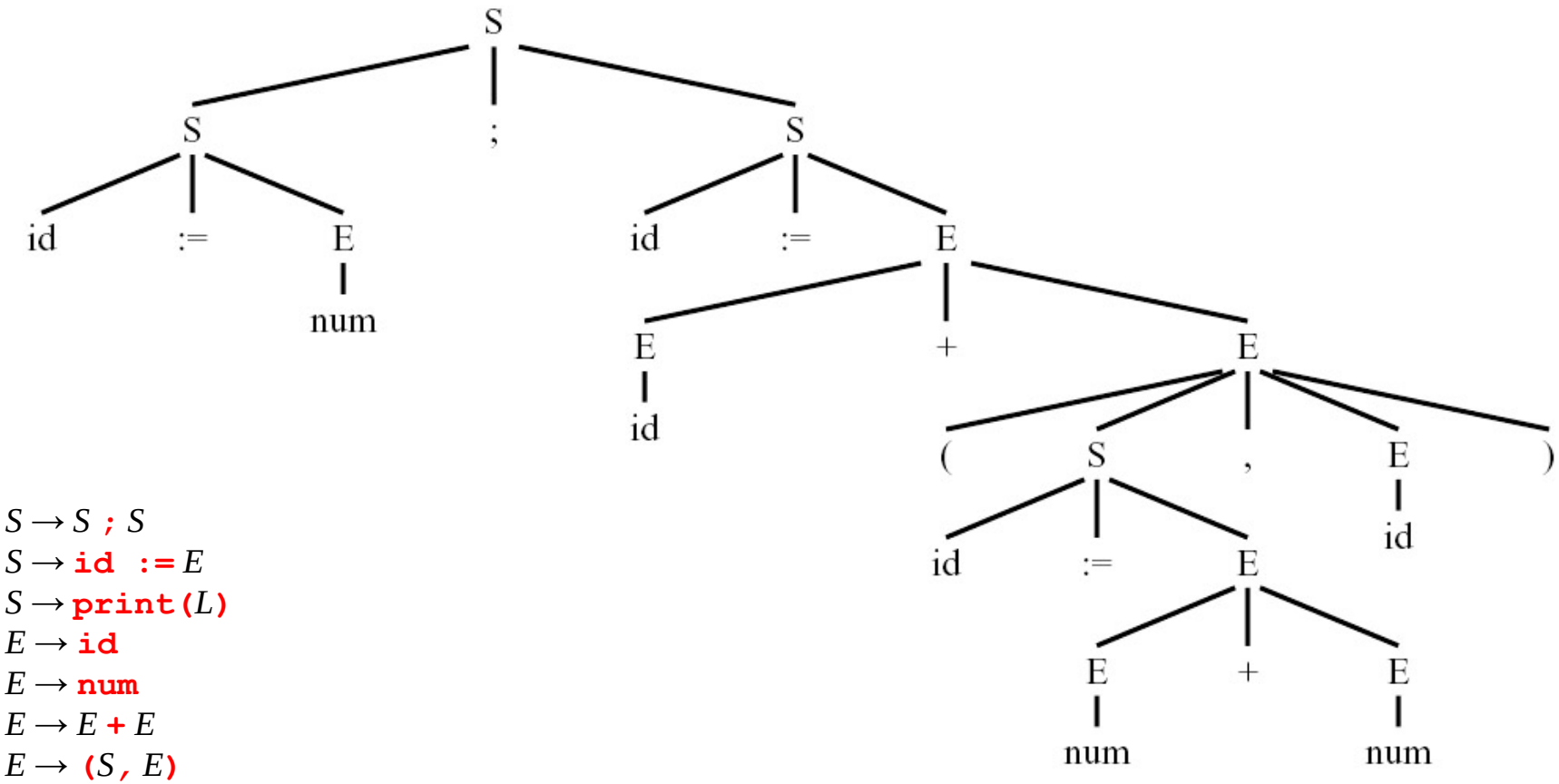
Parse Trees

- Constrói-se uma árvore conectando-se cada símbolo em uma derivação; da qual ele foi derivado.

- Duas derivações diferentes podem levar a uma mesma *parse tree.*

Parse Trees

a : = 7; b : = c + (d : = 5 + 6, d)

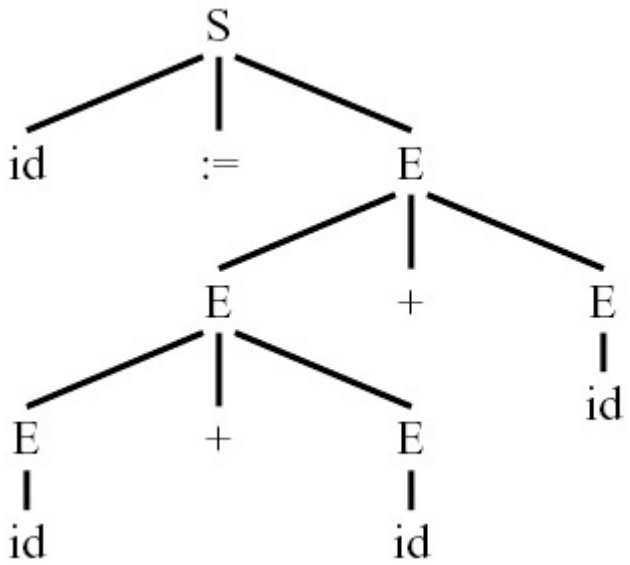


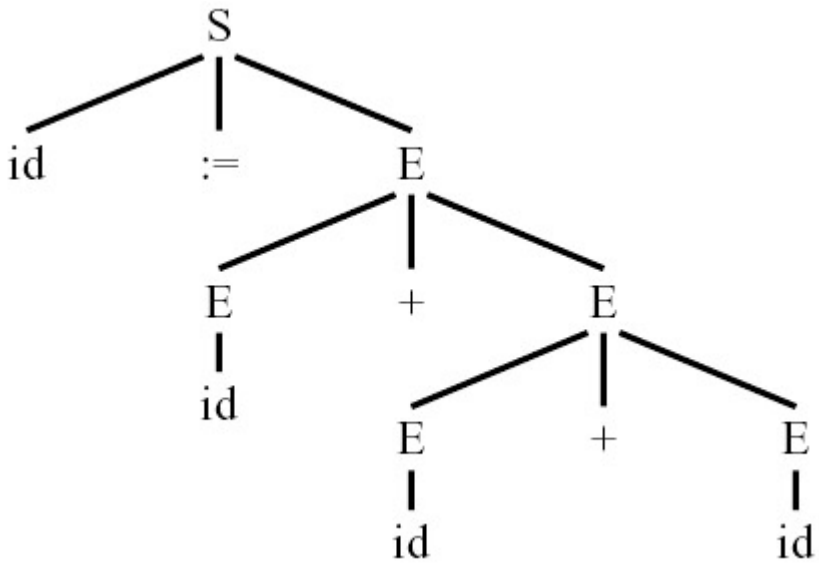
1. $S \rightarrow S$ **;** S
2. $S \rightarrow$ **id :=** E
3. $S \rightarrow$ **print(**L**)**
4. $E \rightarrow$ **id**
5. $E \rightarrow$ **num**
6. $E \rightarrow E$ **+** E
7. $E \rightarrow$ **(**S**,** E**)**
8. $L \rightarrow E$
9. $L \rightarrow L$ **,** E

Gramáticas Ambíguas

Gramáticas Ambíguas: Podem derivar uma sentença com duas *parse trees* diferentes

id := id+id+id





É ambígua?

$E \rightarrow \mathbf{id}$

$E \rightarrow \mathbf{num}$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow E + E$

$E \rightarrow E - E$

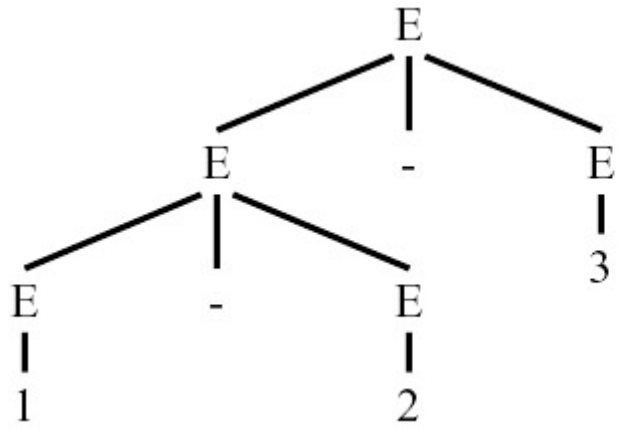
$E \rightarrow (E)$

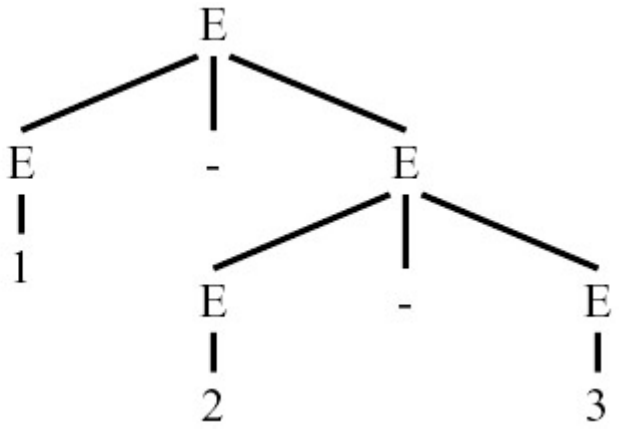
Construa *Parse Trees* para as seguintes expressões:

1-2-3

1+2*3

Exemplo: 1-2-3

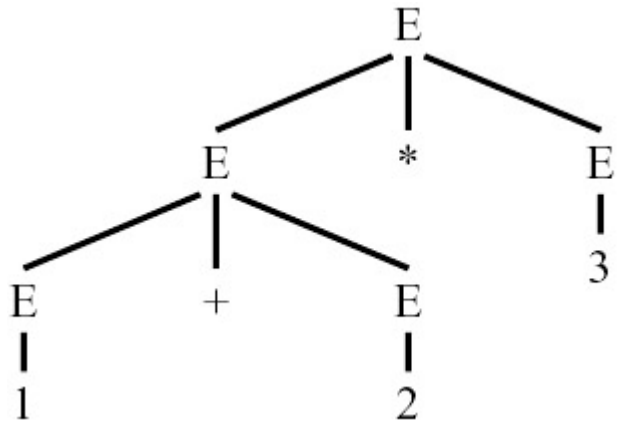


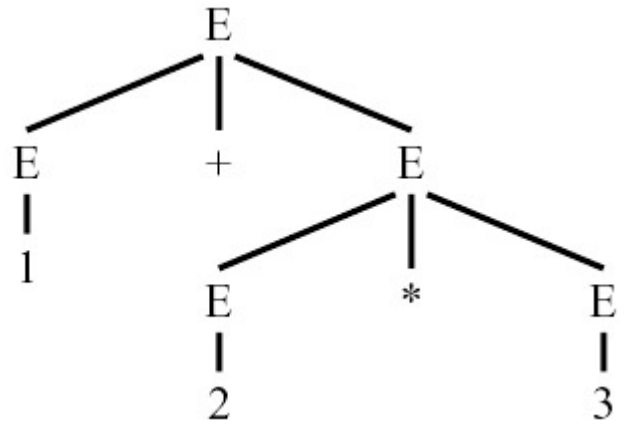


Ambígua!

(1-2)-3 = -4 e 1-(2-3) = 2

Exemplo: 1+2*3





Ambígua!

(1+2)*3 = 9 e 1+(2*3) = 7

Gramáticas Ambíguas

- Gera uma mesma cadeia com duas árvores sintáticas diferentes
- Pode-se formalizar assim:
  - Gramáticas ambíguas geram alguma cadeia ambiguamente
  - Uma cadeia é gerada ambiguamente se possui duas ou mais derivações mais à esquerda diferentes.
- Os compiladores usam as *parse trees* para extrair o significado das expressões
- A ambigüidade se torna um problema
- Pode-se, geralmente, mudar a gramática de maneira a retirar a ambigüidade

Gramáticas Ambíguas

Alterando o exemplo anterior:

- Deseja-se colocar uma precedência maior para * em relação a + e –

- Também deseja-se que cada operador seja associativo à esquerda:

  (1-2)-3 e não 1-(2-3)

Consegue-se isso introduzindo novos não-terminais

Gramáticas para Expressões

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

$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$

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

Construa as derivações e *Parse Trees* para as seguintes expressões:

1-2-3

1+2*3

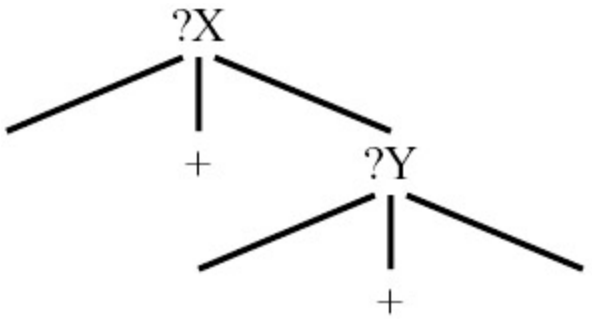
Gramáticas para Expressões

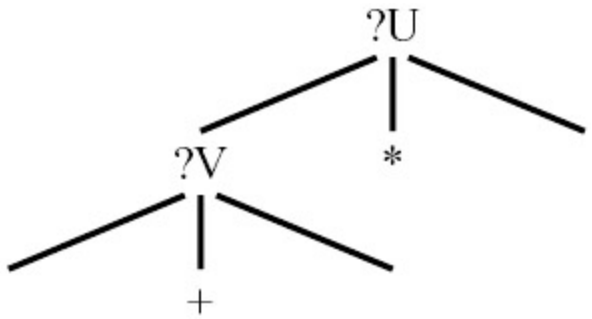
$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$

$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$

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

Essa gramática pode gerar as árvores abaixo?





Gramáticas Ambíguas

- Geralmente pode-se transformar uma gramática para retirar a ambigüidade
- Algumas linguagens não possuem gramáticas não ambíguas
- Mas elas não seriam apropriadas como linguagens de programação

Parsing

CFG's geram as linguagens.

Parsers são reconhecedores das linguagens.

Para qualquer CFG é possível obter um *parser* que roda em $O(n^3)$ $\rightarrow$ Algoritmos de Earley e CYK (Cocke-Younger-Kasami).

$O(n^3)$ é muito lento para programas grandes.

Existem classes de gramáticas para as quais podemos construir *parsers* que rodam em tempo linear. Exemplo:

LL: **L**eft-to-right, **L**eft-most derivation
LR: **L**eft-to-right, **R**ight-most derivation

Lista de Exercícios

Lista 8

- Exercícios teóricos