

Conteúdo Programático e Cronograma

1º Semestre

~~Organização e estrutura de compiladores~~

~~Análise Léxica~~

Análise Sintática

Ferramentas de geração automática de compiladores

2º Semestre

Análise Semântica

Geração de código

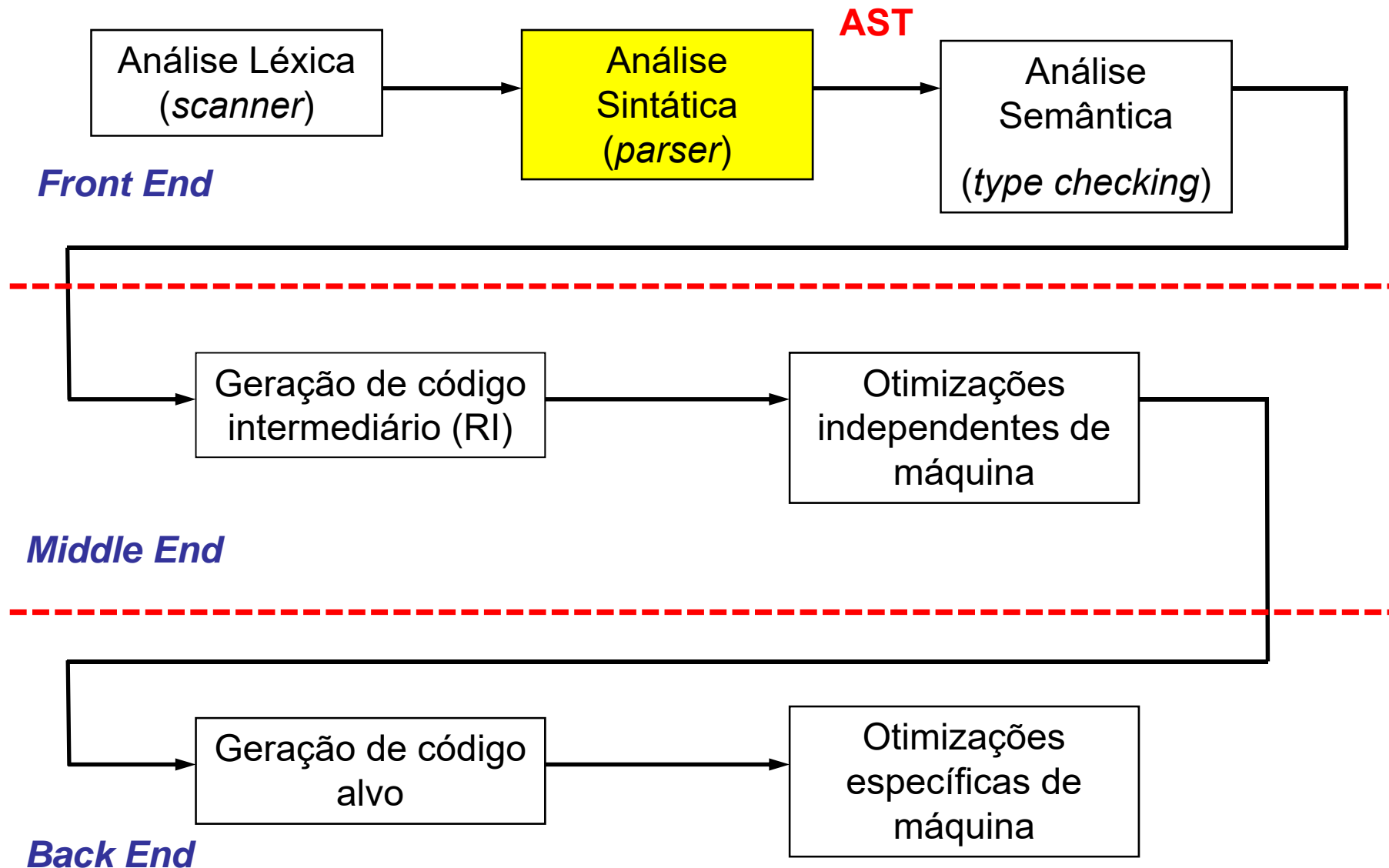
Análise de fluxo de dados

Otimização de código

Alocação de registradores

Análise Sintática

Fluxo do Compilador



Analizador 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.

Analizador Sintático (*Parser*)

- **ERs** 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 ???

Analizador Sintático (*Parser*)

- As **ERs** devem ser capazes de expressar a sintaxe de linguagens de programação.
- E se forem dados nomes para abreviar as **ERs**?

EXPR = ab(c|d)e



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

Analizador Sintático (*Parser*)

- Exemplo de **ER** usando abreviações:

digits = [0-9]⁺

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 = ([0-9]⁺ “+”)^{*} [0-9]⁺

Analizador 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

Analizador 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*:

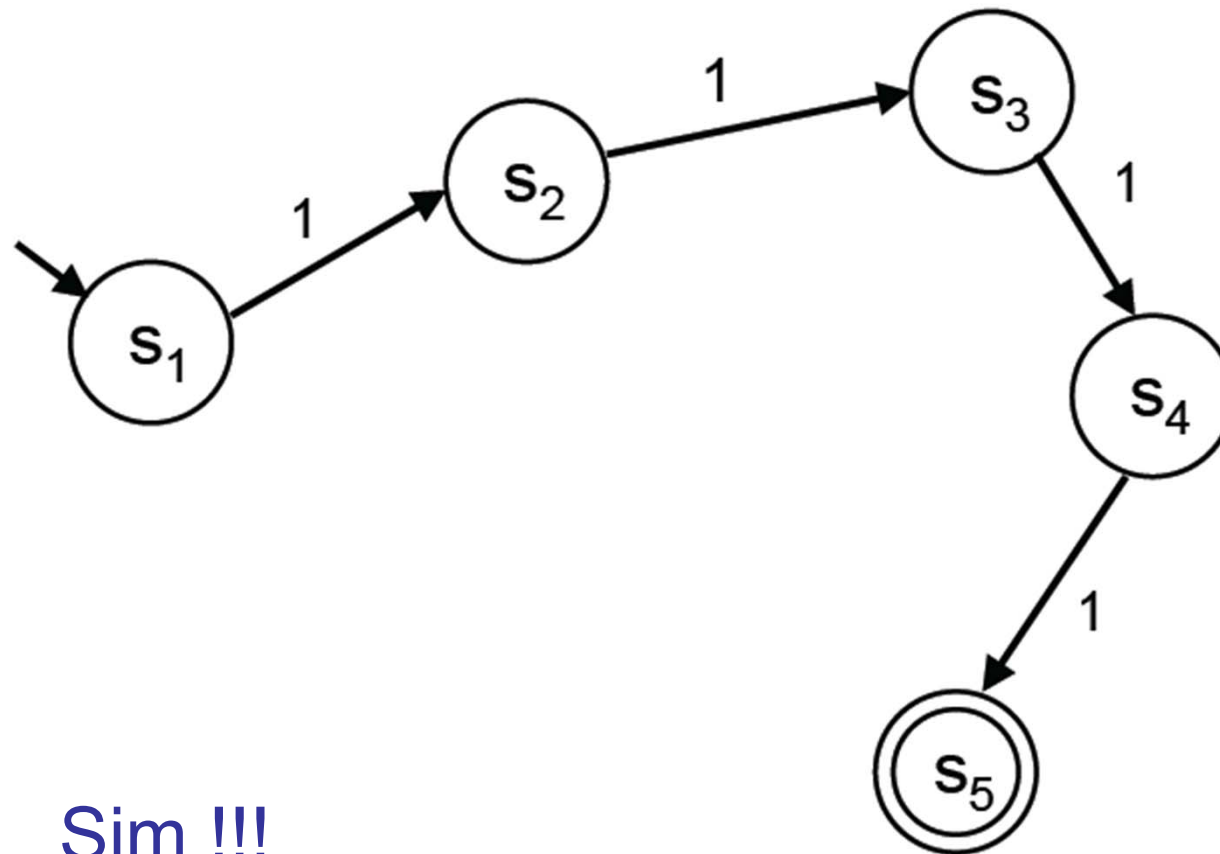
expr = “(” (“(” expr “+” expr “)” | digits) “+” expr “)” | digits

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

Analizador Sintático (*Parser*)

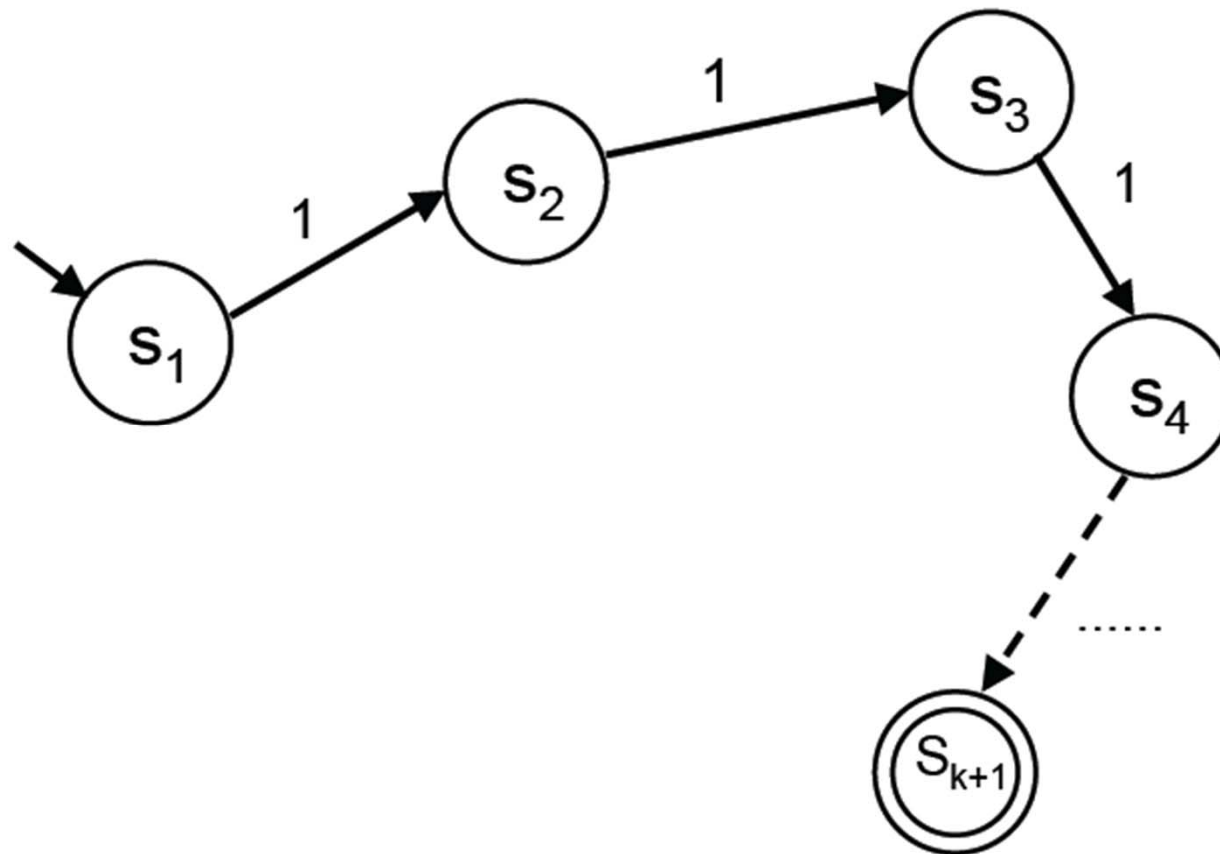
- As abreviações não acrescentam a ERs o poder de expressar recursão.
- É isso que se precisa para expressar a recursão mútua entre `sum` e `expr` e também expressar a sintaxe de linguagens de programação.
- **O que está faltando?**

É possível contar 4 “1”s com um DFA?

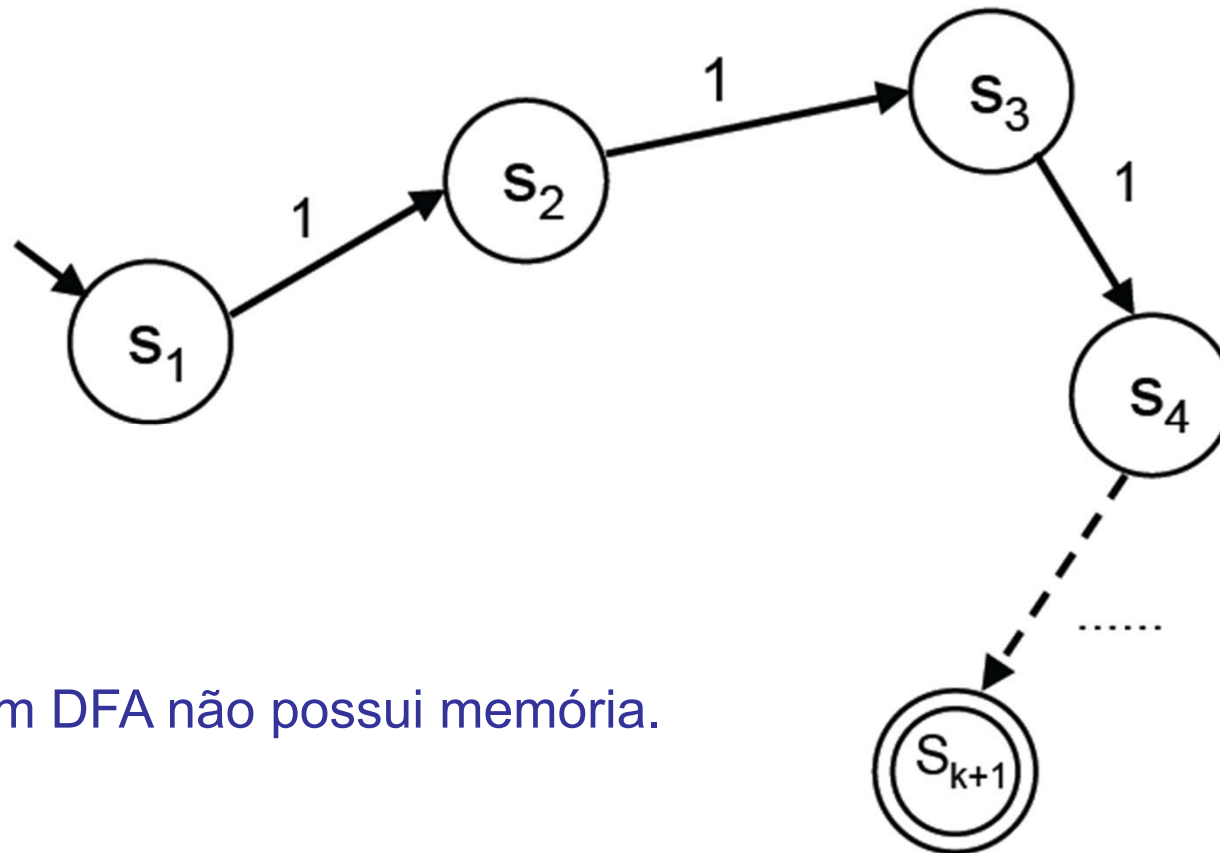


Sim !!!

É possível contar k “1”s com um DFA?

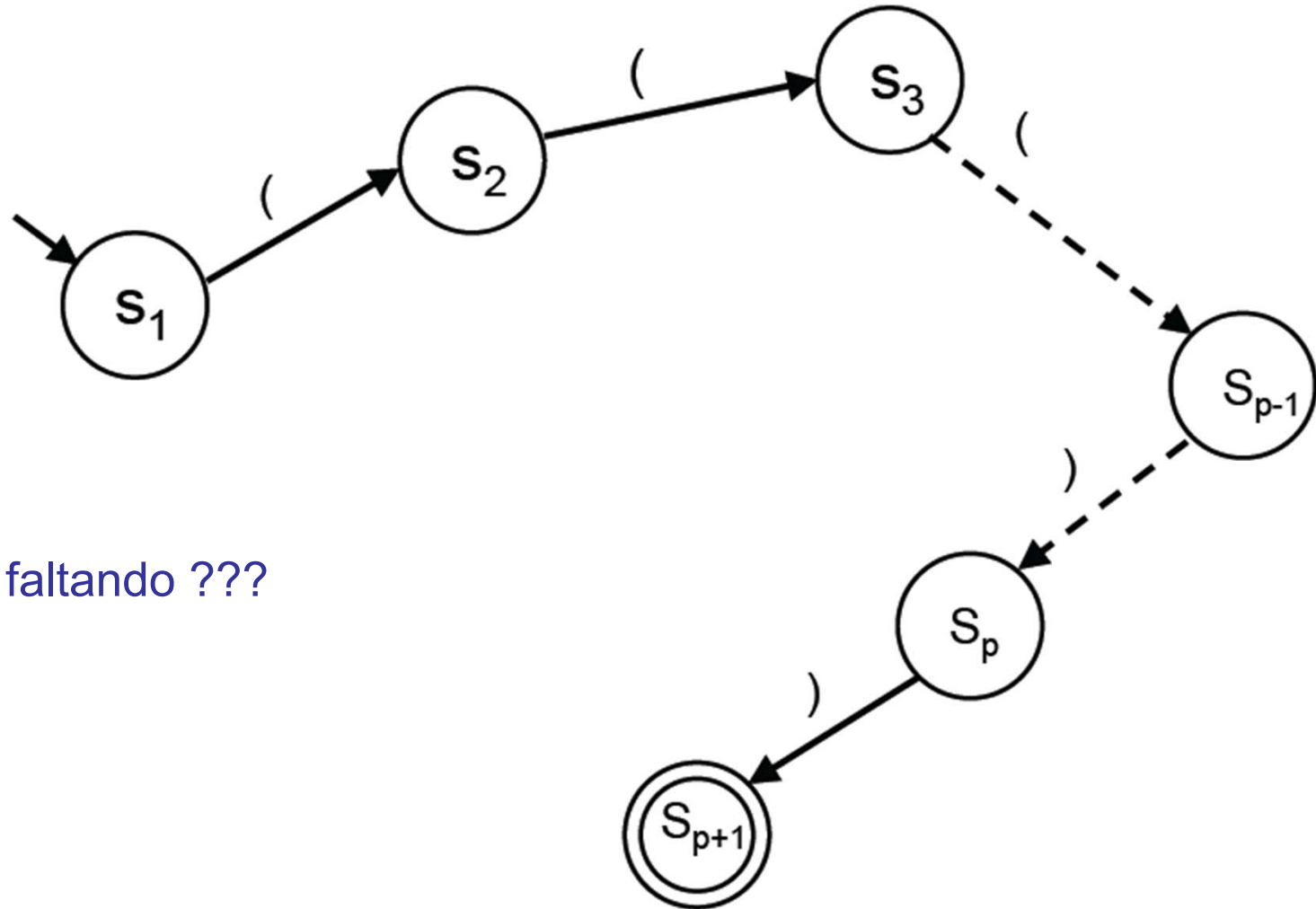


É possível contar k “1”s com um DFA?



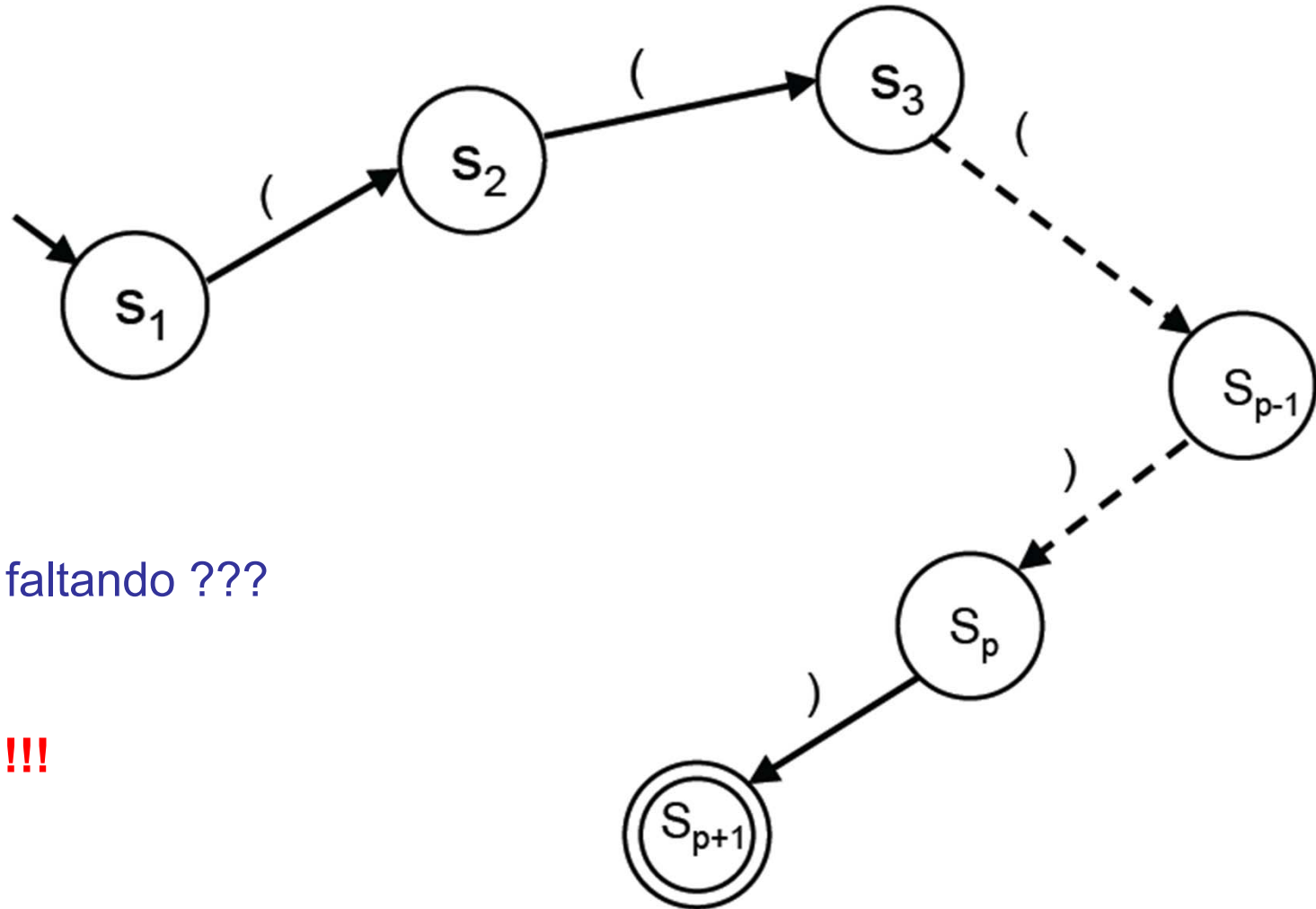
Não, pois um DFA não possui memória.

Como então casar $((\dots))$ para um k qualquer?



O que está faltando ???

Como então casar $((...))$ para um k qualquer?

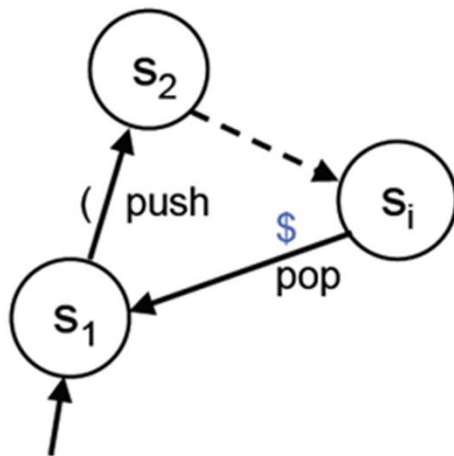


O que está faltando ???

Uma Pilha !!!

Contando com uma Pilha

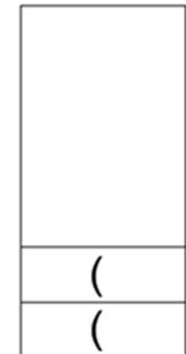
input: ((.....)) \$



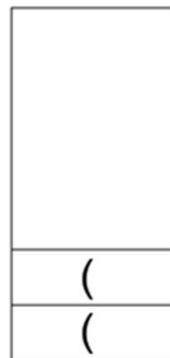
token: (
ação: push (



token: (
ação: push (



...



token:)
ação: pop (



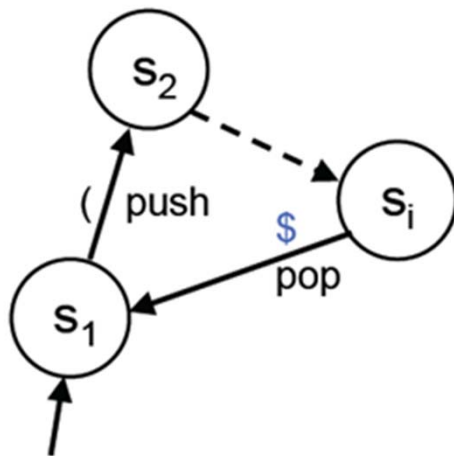
token:)
ação: pop (



termina
vazia

Contando com uma Pilha

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



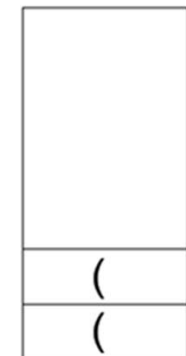
**Uma autômato com
pilha corresponde a que?**



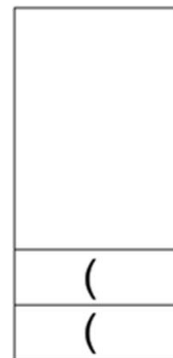
token: (
ação: push (



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:

$$\textit{symbol} \rightarrow \textit{symbol} \textit{symbol} \textit{symbol} \dots \textit{symbol}$$

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

$$1. A \rightarrow 0A1$$

$$2. A \rightarrow B$$

$$3. B \rightarrow \#$$

Context-Free Grammar

A sequência de substituições é chamada de derivação.

Ex:

$A \rightarrow 0A1$

$A \rightarrow B$

$B \rightarrow \#$

- 000#111
- $A \rightarrow 0A1 \rightarrow 00A11 \rightarrow 000A111 \rightarrow 000B111 \rightarrow 000\#111$

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

$$1. S \rightarrow S ; S$$

$$2. S \rightarrow \text{id} := E$$

$$3. S \rightarrow \text{print}(L)$$

$$4. E \rightarrow \text{id}$$

$$5. E \rightarrow \text{num}$$

$$6. E \rightarrow E + E$$

$$7. E \rightarrow (S, E)$$

$$8. L \rightarrow E$$

$$9. L \rightarrow L , E$$

`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)

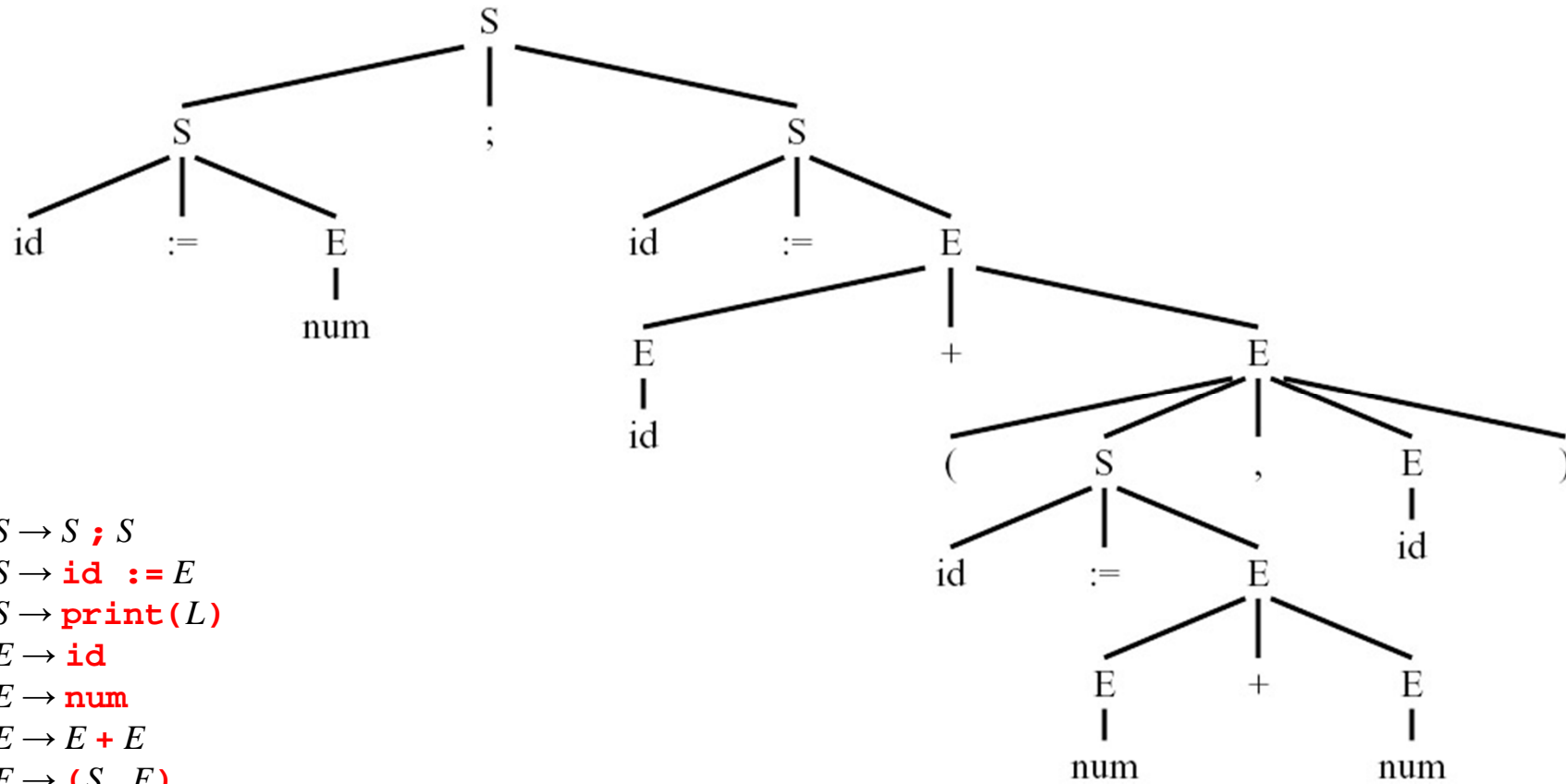
1. $S \rightarrow S ; S$
2. $S \rightarrow \text{id} := E$
3. $S \rightarrow \text{print}(L)$
4. $E \rightarrow \text{id}$
5. $E \rightarrow \text{num}$
6. $E \rightarrow E + E$
7. $E \rightarrow (S, E)$
8. $L \rightarrow E$
9. $L \rightarrow L , E$

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*.

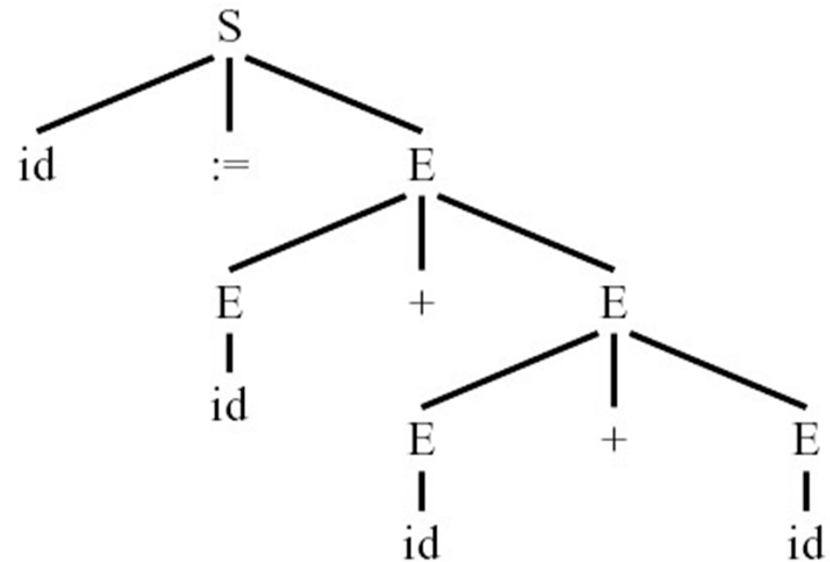
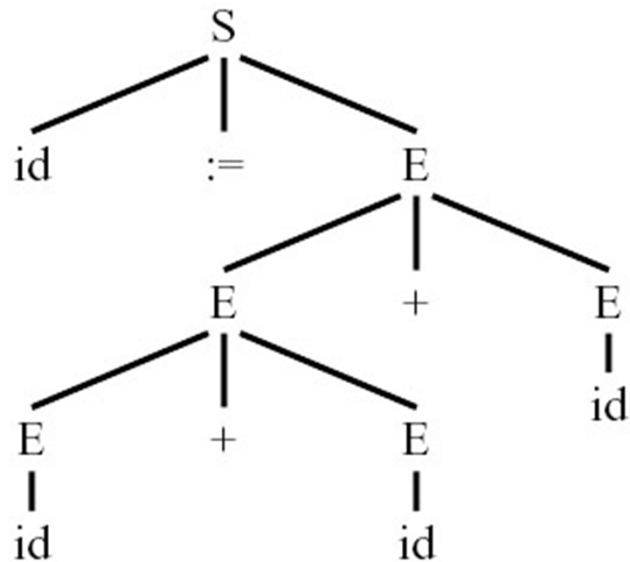


1. $S \rightarrow S ; S$
2. $S \rightarrow \text{id} := E$
3. $S \rightarrow \text{print}(L)$
4. $E \rightarrow \text{id}$
5. $E \rightarrow \text{num}$
6. $E \rightarrow E + E$
7. $E \rightarrow (S, E)$
8. $L \rightarrow E$
9. $L \rightarrow L, E$

Gramáticas Ambíguas

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

id := id+id+id



É ambígua?

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow E + E$

$E \rightarrow E - E$

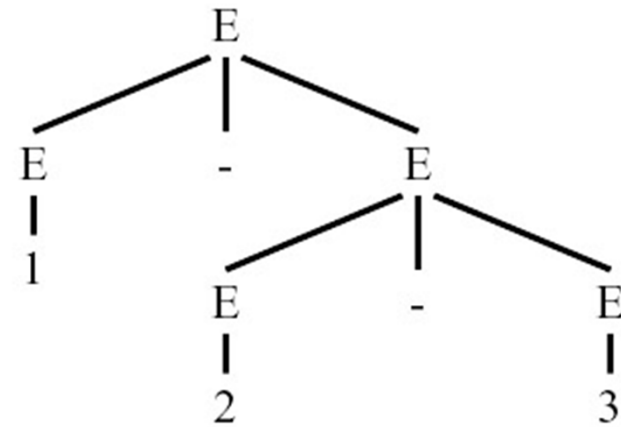
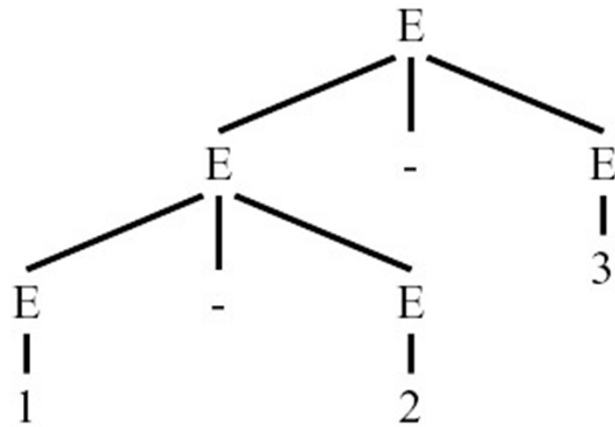
$E \rightarrow (E)$

Construa *Parse Trees* para as seguintes expressões:

1-2-3

1+2*3

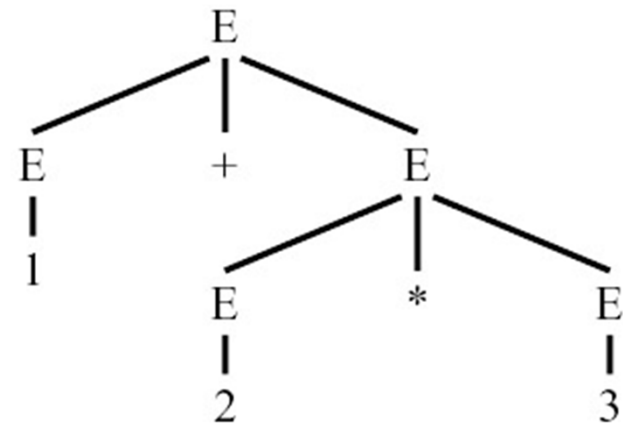
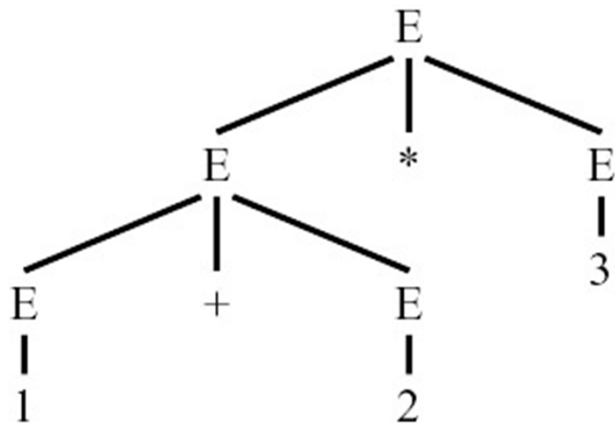
Exemplo: 1-2-3



Ambígua!

$$(1-2)-3 = -4 \quad \text{e} \quad 1-(2-3) = 2$$

Exemplo: $1+2*3$



Ambígua!

$$(1+2)*3 = 9 \quad \text{e} \quad 1+(2*3) = 7$$

Gramáticas Ambíguas

- Gera uma mesma cadeia com duas árvores sintáticas diferentes
- Pode-se formalizar assim:
 - Gramáticas ambíguas geram alguma cadeia ambigualmente
 - Uma cadeia é gerada ambigualmente 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

$$E \rightarrow E + T$$

$$T \rightarrow T * F$$

$$F \rightarrow \text{id}$$

$$E \rightarrow E - T$$

$$T \rightarrow T / F$$

$$F \rightarrow \text{num}$$

$$E \rightarrow T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

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

1-2-3

1+2*3

Gramáticas para Expressões

$$E \rightarrow E + T$$

$$T \rightarrow T * F$$

$$F \rightarrow \text{id}$$

$$E \rightarrow E - T$$

$$T \rightarrow T / F$$

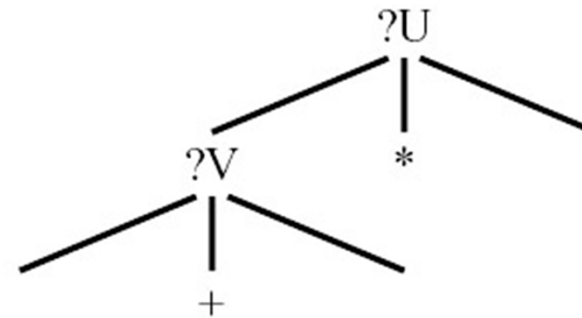
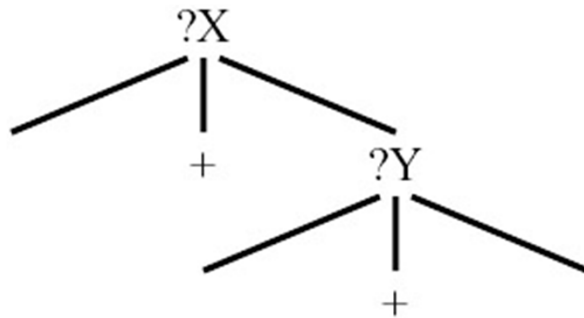
$$F \rightarrow \text{num}$$

$$E \rightarrow T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

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 $O(n^3)$ → Algoritmos de Early[70] 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

Análise Descendente (Predictive Parsing)

Também chamada de *recursive-descent* ou *top-down*

É um algoritmo simples, capaz de fazer o *parsing* de gramáticas LL

Cada produção se torna uma cláusula em uma função recursiva

Tem-se uma função para cada não-terminal

A análise descendente produz uma derivação à esquerda

Ela precisa determinar a produção a ser usada para expandir o não-terminal corrente

Análise Descendente (Predictive Parsing)

$$E \rightarrow +EE$$

$$E \rightarrow *EE$$

$$E \rightarrow a$$

$$E \rightarrow b$$

Expressões pré-fixas

Considere a cadeia **+b*ab**

Como é sua derivação mais à esquerda?

Análise Descendente (Predictive Parsing)

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S L$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; S L$

$E \rightarrow \text{num} = \text{num}$

Como seria um *parser*
para essa gramática?

Análise Descendente (Predictive Parsing)

```
int IF=1, THEN=2, ELSE=3, BEGIN=4, END=5, PRINT=6, SEMI=7, NUM=8, EQ=9;
```

```
int token = getToken();
```

```
void advance() {token=getToken();}
```

```
void eat(int t) {if (token==t) advance(); else error();}
```

```
void S(){
```

```
    switch(token) {
```

```
        case IF: eat(IF); E(); eat(THEN); S(); eat(ELSE); S(); break;
```

```
        case BEGIN: eat(BEGIN); S(); L(); break;
```

```
        case PRINT: eat(PRINT); E(); break;
```

```
        default: error(); }
```

```
}
```

```
void L(){
```

```
    switch(token) {
```

```
        case END: eat(END); break;
```

```
        case SEMI: eat(SEMI); S(); L(); break;
```

```
        default: error(); }
```

```
}
```

```
void E(){ eat(NUM); eat(EQ); eat(NUM); }
```

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S L$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; S L$

$E \rightarrow \text{num} = \text{num}$

Fim de Arquivo

Criar um novo não terminal como símbolo inicial

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

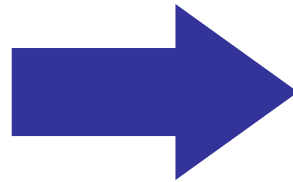
$$T \rightarrow T / F$$

$$T \rightarrow F$$

$$F \rightarrow \text{id}$$

$$F \rightarrow \text{num}$$

$$F \rightarrow (E)$$



$$S \rightarrow E \$$$

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow T / F$$

$$T \rightarrow F$$

$$F \rightarrow \text{id}$$

$$F \rightarrow \text{num}$$

$$F \rightarrow (E)$$

Análise Descendente (Predictive Parsing)

$$S \rightarrow E \$$$

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow T / F$$

$$T \rightarrow F$$

$$F \rightarrow \text{id}$$

$$F \rightarrow \text{num}$$

$$F \rightarrow (E)$$

Vamos aplicar a mesma técnica para essa outra gramática ...

Análise Descendente (Predictive Parsing)

```
void S() { E(); eat(EOF); }
```

```
void E() {  
    switch (tok) {  
        case ?: E(); eat(PLUS); T(); break;  
        case ?: E(); eat(MINUS); T(); break;  
        case ?: T(); break;  
        default: error(); }  
}
```

```
void T() {  
    switch (tok) {  
        case ?: T(); eat(TIMES); F(); break;  
        case ?: T(); eat(DIV); F(); break;  
        case ?: F(); break;  
        default: error(); }  
}
```

Funciona ???

- Como seria a execução para $1*2-3+4$?
- E para $1*2-3$?

$S \rightarrow E \$$
 $E \rightarrow E + T$
 $E \rightarrow E - T$
 $E \rightarrow T$
 $T \rightarrow T * F$
 $T \rightarrow T / F$
 $T \rightarrow F$
 $F \rightarrow \text{id}$
 $F \rightarrow \text{num}$
 $F \rightarrow (E)$

Análise Descendente (Predictive Parsing)

Como decidir entre $E+T$, $E-T$ e T na função que implementa o não-terminal E ?

- Tanto E como T podem derivar cadeias começando com **id**, **num** ou **(**
- E se fosse possível olhar um número $k > 1$ de símbolos para frente na entrada?

Análise Descendente (Predictive Parsing)

Como decidir entre $E+T$, $E-T$ e T na função que implementa o não-terminal E ?

- Tanto E como T podem derivar cadeias começando com **id**, **num** ou **(**
- E se fosse possível olhar um número $k > 1$ de símbolos para frente na entrada?

Essas cadeias podem ter tamanho arbitrário: O problema permanece

Análise descendente recursiva (preditiva) só funciona onde o primeiro símbolo terminal de cada sub-expressão permite escolher a produção adequada a ser utilizada na derivação

Análise Descendente

$S \rightarrow E \$$

$E \rightarrow F + T$

$E \rightarrow G + T$

$F \rightarrow \text{id}$

$G \rightarrow \text{num}$

$T \rightarrow \text{num}$

Como seria a análise da cadeia $\text{num+num\$}$?

Análise Descendente LL(1)

$S \rightarrow E \$$

$E \rightarrow F + T$

$E \rightarrow G + T$

$F \rightarrow \text{id}$

$G \rightarrow \text{num}$

$T \rightarrow \text{num}$

	id	num	+	\$
S	$S \rightarrow E \$$	$S \rightarrow E \$$		
E	$E \rightarrow F + T$	$E \rightarrow G + T$		
F	$F \rightarrow \text{id}$			
G		$G \rightarrow \text{num}$		
T		$T \rightarrow \text{num}$		

Como seria a análise da cadeia **num+num\$** ?

Conjunto FIRST

- Dada uma cadeia γ de terminais e não terminais $\text{FIRST}(\gamma)$ é o conjunto de todos os terminais que podem iniciar uma cadeia derivada de γ .

- Exemplo usando gramática ao lado:

$$\gamma = T * F$$

$$\text{FIRST}(\gamma) = \{ \text{id}, \text{num}, (\}$$

$S \rightarrow E \$$
$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$
$F \rightarrow \text{id}$
$F \rightarrow \text{num}$
$F \rightarrow (E)$

Predictive Parsing

Se uma gramática tem produções da forma:

$$X \rightarrow \gamma_1$$

$$X \rightarrow \gamma_2$$

- Caso os conjuntos $\text{FIRST}(\gamma_1)$ e $\text{FIRST}(\gamma_2)$ tenham intersecção, então a gramática não pode ser analisada com um *predictive parser*

Por que?

A função recursiva não vai saber que caso executar

Calculando FIRST

$Z \rightarrow d$

$Z \rightarrow X Y Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

- Como seria para $\gamma = XYZ$?

- Pode-se simplesmente fazer
 $\text{FIRST}(XYZ) = \text{FIRST}(X)$?

Nullable

Nullable(X) é verdadeiro se X pode derivar a cadeia vazia.

$Z \rightarrow d$

$Z \rightarrow X Y Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

Nullable(Y) = yes

Nullable(X) = yes

Nullable(Z) = no

Follow

$\text{FOLLOW}(X)$ é o conjunto de terminais que podem imediatamente seguir X

$t \in \text{FOLLOW}(X)$ se existe alguma derivação contendo Xt

Cuidado com derivações da forma $XYZt$, onde Y e Z podem ser vazios

$$Z \rightarrow d$$

$$Z \rightarrow X Y Z$$

$$Y \rightarrow$$

$$Y \rightarrow c$$

$$X \rightarrow Y$$

$$X \rightarrow a$$

$$\text{FOLLOW}(Y) = \{d, a, c\}$$

$$\text{FOLLOW}(Z) = \{ \}$$

FIRST, FOLLOW e Nullable

- $\text{Nullable}(X)$ é verdadeiro se X pode derivar a cadeia vazia
- $\text{FIRST}(\gamma)$ é o conjunto de terminais que podem iniciar cadeias derivadas de γ
- $\text{FOLLOW}(X)$ é o conjunto de terminais que podem imediatamente seguir X
 - $t \in \text{FOLLOW}(X)$ se existe alguma derivação contendo Xt
 - Cuidado com derivações da forma $XYZt$, onde Y e Z podem ser vazios

FIRST, FOLLOW e Nullable

Initialize FIRST and FOLLOW to all empty sets, and Nullable to all false.

```
for each terminal symbol  $Z$   $\text{FIRST}[Z] \leftarrow \{Z\}$ 
repeat
  for each production  $X \rightarrow Y_1 Y_2 \dots Y_k$ 
    if  $Y_1 \dots Y_k$  are all Nullable (or if  $k = 0$ ) then  $\text{Nullable}[X] \leftarrow \text{true}$ 
    for each  $i$  from 1 to  $k$ , each  $j$  from  $i + 1$  to  $k$ 
      if  $Y_1 \dots Y_{i-1}$  are all Nullable (or if  $i = 1$ )
        then  $\text{FIRST}[X] \leftarrow \text{FIRST}[X] \cup \text{FIRST}[Y_i]$ 
      if  $Y_{i+1} \dots Y_k$  are all Nullable (or if  $i = k$ )
        then  $\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FOLLOW}[X]$ 
      if  $Y_{i+1} \dots Y_{j-1}$  are all Nullable (or if  $i + 1 = j$ )
        then  $\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FIRST}[Y_j]$ 
  until FIRST, FOLLOW, and Nullable did not change in this iteration.
```

Generalizando para cadeias: FIRST

- $\text{FIRST}(X \gamma) = \text{FIRST}[X]$, if not nullable[X]
- $\text{FIRST}(X \gamma) = \text{FIRST}[X] \cup \text{FIRST}(\gamma)$, if nullable[X]
- A cadeia γ é *Nullable* se cada símbolo em γ é *Nullable*

Generalizando para cadeias: FOLLOW

- Se houver uma produção $A \rightarrow \alpha B \beta$, então, tudo em $\text{FIRST}(\beta)$ irá para $\text{FOLLOW}(B)$
- Se houver uma produção $A \rightarrow \alpha B$, ou uma produção $A \rightarrow \alpha B \beta$ onde $\text{FIRST}(\beta)$ é Nullable, então tudo em $\text{FOLLOW}(A)$ irá para $\text{FOLLOW}(B)$

Exemplo

$Z \rightarrow d$

$Z \rightarrow X Y Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

	nullable	FIRST	FOLLOW
X	no		
Y	no		
Z	no		

Exemplo

$Z \rightarrow d$

$Z \rightarrow X Y Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

	nullable	FIRST	FOLLOW
X	yes	a c	a c d
Y	yes	c	a c d
Z	no	a c d	

Construindo um Predictive Parser LL(1)

- Cada função relativa a um não-terminal precisa conter uma cláusula para cada produção
- A escolha da produção adequada é baseada no próximo *token*
- Isto é feito através da *predictive parsing table*
- Dada uma produção $X \rightarrow \gamma$
- Para cada terminal $T \in \text{FIRST}(\gamma)$
 - Coloque a produção $X \rightarrow \gamma$ na linha X , coluna T .
- Se γ é *nullable*:
 - Coloque a produção na linha X , coluna T para cada $T \in \text{FOLLOW}[X]$.

Exemplo

		nullable	FIRST	FOLLOW
$Z \rightarrow d$	X	yes	a c	a c d
	Y	yes	c	a c d
$Z \rightarrow X Y Z$	Z	no	a c d	
$Y \rightarrow$				
$Y \rightarrow c$				
$X \rightarrow Y$				
$X \rightarrow a$				

$$X \rightarrow a$$

Funciona ???

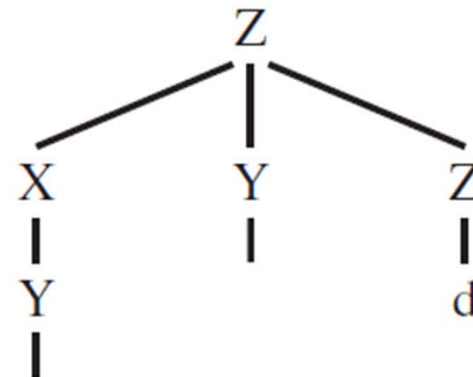
213

Construindo um Predictive Parser LL(1)

Não Funciona!! Por quê?

- A gramática é ambígua
- Note que algumas células da tabela do *predictive parser* têm mais de uma entrada!
- Isso sempre acontece com gramáticas ambíguas!

Z
|
d



Construindo um Predictive Parser LL(1)

- Linguagens cujas tabelas não possuam entradas duplicadas são denominadas de LL(1)
 - *Left to right parsing, leftmost derivation, 1-symbol lookahead*
- A definição de conjuntos FIRST pode ser generalizada para os primeiros k tokens de uma string
 - Gera-se uma tabela onde as linhas são os não-terminais e as colunas são todas as seqüências possíveis de k terminais
- Isso é raramente feito devido ao tamanho explosivo das tabelas geradas
- Gramáticas analisáveis com tabelas LL(k) são chamadas LL(k)
- Nenhuma gramática ambígua é LL(k) para nenhum k !

Exemplo: Gramática LL(1)

$$E \rightarrow TE'$$

$$E' \rightarrow +TE'$$

$$E' \rightarrow$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'$$

$$T' \rightarrow$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

Exemplo: Gramática LL(1)

$S \rightarrow E\$$ ← inserção do fim de arquivo

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T' \rightarrow$

$F \rightarrow (E)$

$F \rightarrow id$

Exemplo: Gramática LL(1)

$S \rightarrow E\$$

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T' \rightarrow$

$F \rightarrow (E)$

$F \rightarrow id$

	Nullable	FIRST	FOLLOW
E	N	(id) \$
E'	S	+) \$
T	N	(id	+) \$
T'	S	*	+) \$
F	N	(id	* +) \$
S	N	(id	

Exemplo: Gramática LL(1)

$S \rightarrow E\$$

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$E' \rightarrow$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$T' \rightarrow$

$F \rightarrow (E)$

$F \rightarrow id$

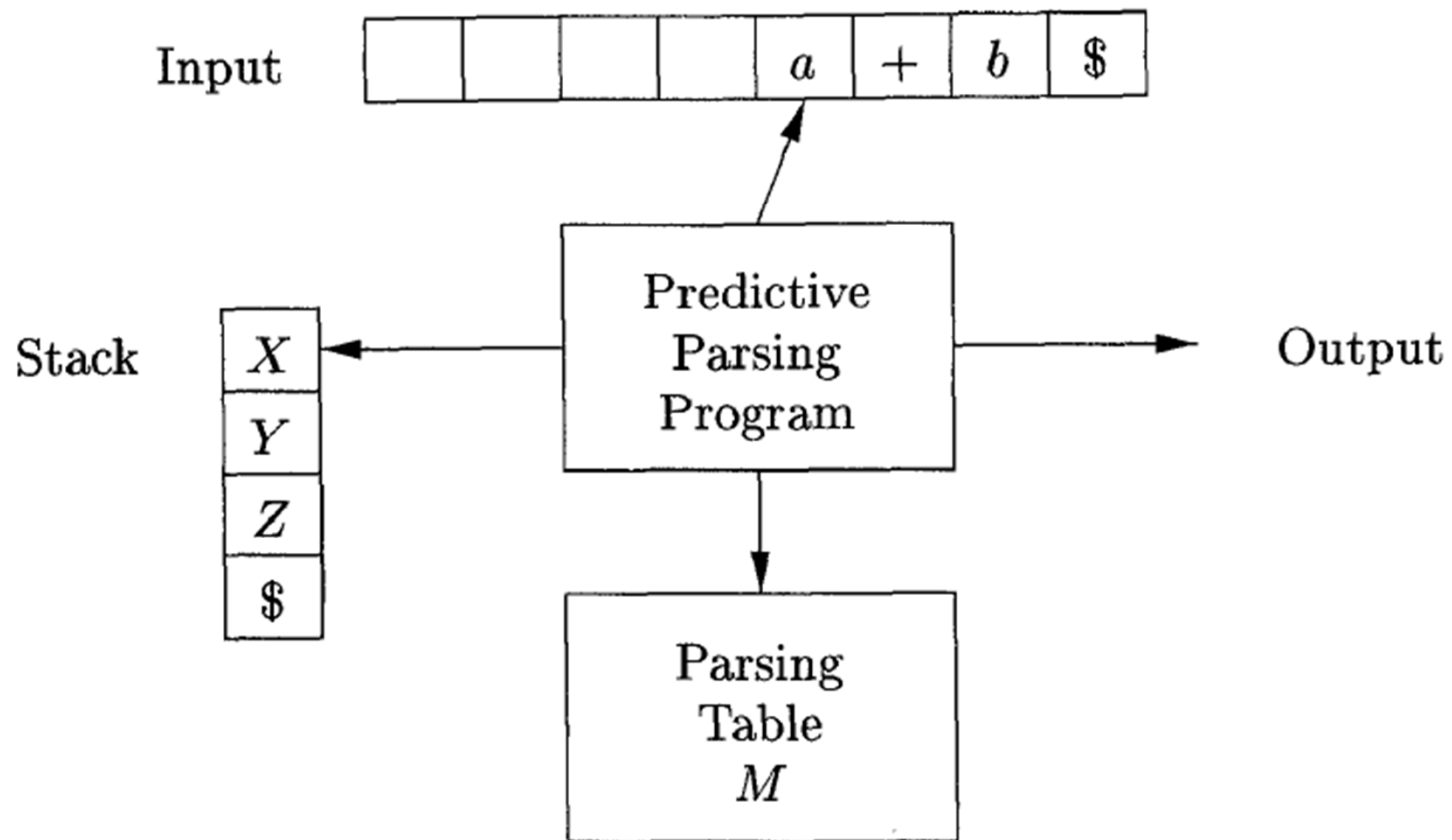
Nullable FIRST FOLLOW

E	N	(id) \$
E'	S	+) \$
T	N	(id	+) \$
T'	S	*	+) \$
F	N	(id	* +) \$
S	N	(id	

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
<i>E</i>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<i>E'</i>		$E' \rightarrow +TE'$			$E' \rightarrow$	$E' \rightarrow$
<i>T</i>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<i>T'</i>		$T' \rightarrow$	$T' \rightarrow *FT'$		$T' \rightarrow$	$T' \rightarrow$
<i>F</i>	$F \rightarrow id$			$F \rightarrow (E)$		
<i>S</i>	$S \rightarrow E\$$			$S \rightarrow E\$$		

Análise Sintática LL(1)

TOP-DOWN PARSING



Análise Sintática LL(1)

$$S \rightarrow E\$$$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE'$$

$$E' \rightarrow$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'$$

$$T' \rightarrow$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

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

A cadeia abaixo pertence a linguagem gerada pela gramática?

id+id*id\$

Exemplo: Gramática LL(1)

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow$	$E' \rightarrow$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow$	$T' \rightarrow *FT'$		$T' \rightarrow$	$T' \rightarrow$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		
S	$S \rightarrow ES$			$S \rightarrow ES$		

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\text{id} + \text{id} * \text{id}\$$	
	$TE'\$$	$\text{id} + \text{id} * \text{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\text{id} + \text{id} * \text{id}\$$	output $T \rightarrow FT'$
	$\text{id} T'E'\$$	$\text{id} + \text{id} * \text{id}\$$	output $F \rightarrow \text{id}$
id	$T'E'\$$	$+ \text{id} * \text{id}\$$	match id
id	$E'\$$	$+ \text{id} * \text{id}\$$	output $T' \rightarrow \epsilon$
id	$+ TE'\$$	$+ \text{id} * \text{id}\$$	output $E' \rightarrow + TE'$
id +	$TE'\$$	$\text{id} * \text{id}\$$	match +
id +	$FT'E'\$$	$\text{id} * \text{id}\$$	output $T \rightarrow FT'$
id +	$\text{id} T'E'\$$	$\text{id} * \text{id}\$$	output $F \rightarrow \text{id}$
id + id	$T'E'\$$	$* \text{id}\$$	match id
id + id	$* FT'E'\$$	$* \text{id}\$$	output $T' \rightarrow * FT'$
id + id *	$FT'E'\$$	$\text{id}\$$	match *
id + id *	$\text{id} T'E'\$$	$\text{id}\$$	output $F \rightarrow \text{id}$
id + id * id	$T'E'\$$	$\$$	match id
id + id * id	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
id + id * id	$\$$	$\$$	output $E' \rightarrow \epsilon$