

---

# Análise Sintática

**Sandro Rigo**  
**sandro@ic.unicamp.br**

# Introdução

---

- **Análise Léxica:**
  - Quebra a entrada em palavras conhecidas como *tokens*
- **Análise Sintática:**
  - Analisa a estrutura de frases do programa
- **Análise Semântica:**
  - Calcula o “significado” do programa

# 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

# Introdução

---

- ERs são boas para definir a estrutura léxica de maneira declarativa
- Não são “poderosas” o suficiente para conseguir definir declarativamente a estrutura sintática de linguagens de programação

# Introdução

---

- 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 =  $([0-9]^+ \text{ “+” } )^* [0-9]^+$

# Introdução

---

- É possível usar a mesma idéia para definir uma linguagem para expressões que tenham parênteses balanceados?
  - $(1+(245+2))$
- Tentativa:
  - $\text{digits} = [0-9]^+$
  - $\text{sum} = \text{expr} \text{ “+” } \text{expr}$
  - $\text{expr} = \text{“(” sum “)”} \mid \text{digits}$

# Introdução

---

- O analisador léxico substituiria *sum* em *expr*:
  - $\text{expr} = \text{"(" expr "+" expr \text{"})"} \mid \text{digits}$
- Depois substituiria *expr* no próprio *expr*:
  - $\text{expr} = \text{"( ( " expr "+" expr \text{"} ) "+" expr \text{"} )"} \mid \text{digits}$
- Continua tendo *expr*'s do lado direito!

# Introdução

---

- Não é possível pois a linguagem de parênteses balanceados ( $L$ ) não é regular
- Idéia da prova:
  - Teorema do bombeamento para LR:

Seja  $L$  uma LR. Então existe inteiro  $p \geq 1$ , dependendo de  $L$ , tal que toda string  $w$  in  $L$  de tamanho  $\geq p$  pode ser escrita como  $w = xyz$ , satisfazendo:

- $|y| \geq 1$
- $|xy| \leq p$
- for all  $i \geq 0$ ,  $xy^iz \in L$



# Introdução

---

- A prova pode ser feita por contradição
  - Assuma que  $L$  seja regular
  - Mostre que uma string que está em  $L$  não satisfaz o teorema
  - Para nosso caso, escolha uma string com um número maior que  $p$  de “(“ no início.

# Gramáticas Livre de Contexto

---

- As abreviações não acrescentam a ERs o poder de expressar recursão.
- É isso que precisamos para expressar a recursão mútua entre sum e expr
- E também para expressar a sintaxe de linguagens de programação

$\text{expr} = ab(c|d)e$



$\text{aux} = c \mid d$

$\text{expr} = a b \text{ aux } e$

# Gramáticas Livre de Contexto

---

- Descreve uma linguagem através de um conjunto de produções da forma:

*symbol*  $\rightarrow$  *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 (variáveis)

# Gramáticas Livre de Contexto

---

- Símbolos:
  - nenhum terminal aparece do lado esquerdo de uma produção
  - existe um não-terminal definido como start symbol
    - Normalmente é o da primeira regra

# Gramáticas Livre de Contexto

---

1.  $A \rightarrow 0A1$
2.  $A \rightarrow B$
3.  $B \rightarrow \#$

- 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

# Gramáticas Livre de Contexto

---

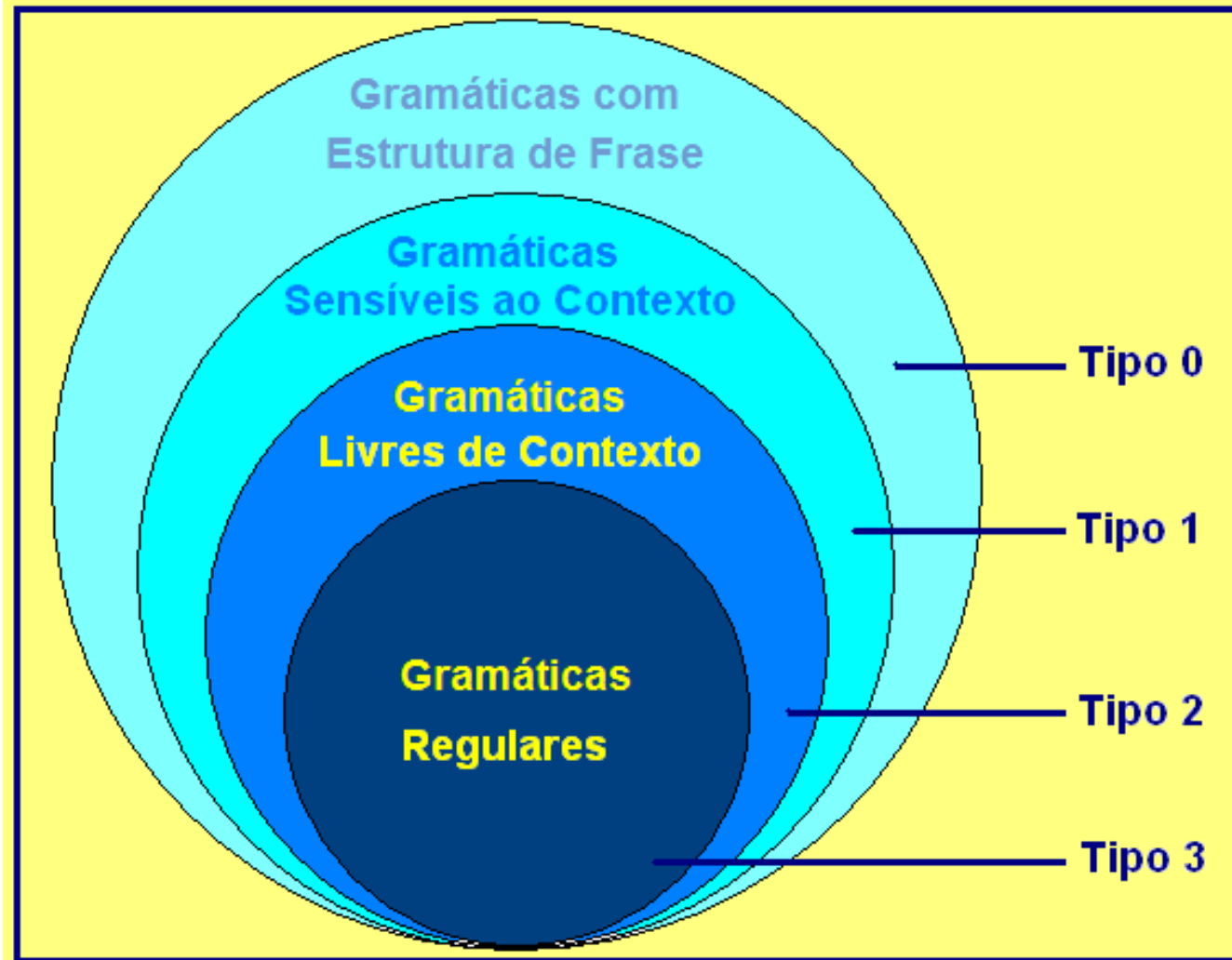
- A sequência de substituições é chamada de derivação
- Ex:
  - 000#111
  - $A \rightarrow 0A1 \rightarrow 00A11 \rightarrow \dots$
- Linguagem: O conjunto de todas as cadeias que podem ser geradas dessa maneira

# Hierarquia de Chomsky

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	<u>Recursively enumerable</u>	<u>Turing machine</u>	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	<u>Context-sensitive</u>	<u>Linear-bounded non-deterministic Turing machine</u>	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	<u>Context-free</u>	Non-deterministic <u>push down automaton</u>	$A \rightarrow \gamma$
Type-3	<u>Regular</u>	<u>Finite state automaton</u>	$A \rightarrow a$ and $A \rightarrow aB$

$A$  e  $B$  non-terminals e  $\alpha\beta\gamma$  strings of terminals and non-terminals,  $\gamma$  non empty

# Hierarquia de Chomsky





# Gramáticas Livre de Contexto

---

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

# Gramáticas Livre de Contexto

---

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

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

- $\underline{S}$
- $S; \underline{S}$
- $\underline{S}; id := E$
- $id := \underline{E}; id := E$
- $id := num; id := \underline{E}$
- $id := num; id := \underline{E} + E$
- $id := num; id := \underline{E} + (S, E)$
- $id := num; id := id + (\underline{S}, E)$
- $id := num; id := id + (id := \underline{E}, E)$
- $id := num; id := id + (id := E + E, \underline{E})$
- $id := num; id := id + (id := \underline{E} + E, id)$
- $id := num; id := id + (id := num + \underline{E}, id)$
- $id := num; id := id + (id := num + num, id)$

# 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?

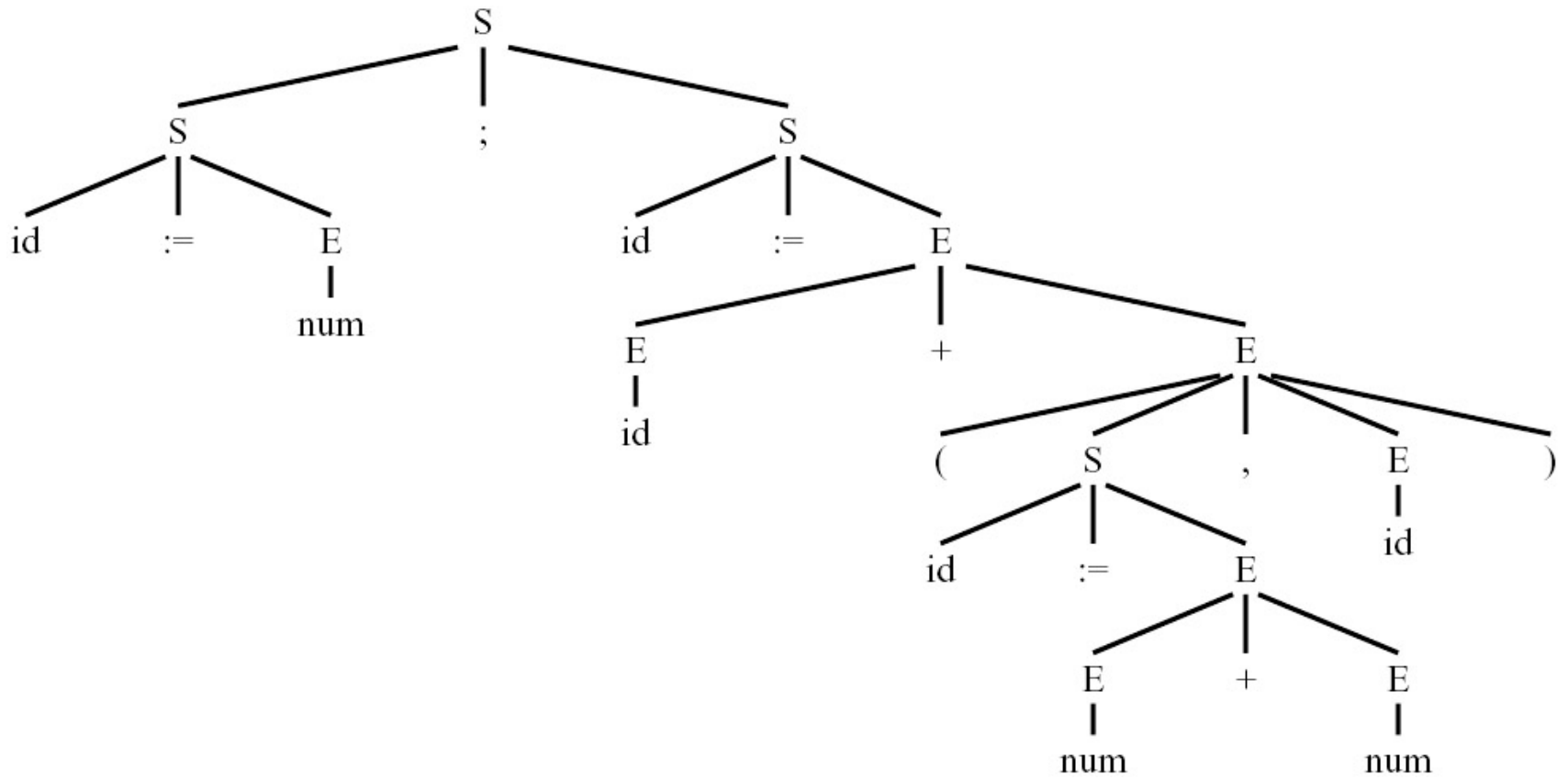
# Parse Trees

---

- Constrói-se uma árvore conectando-se cada símbolo em uma derivação ao qual ele foi derivado
- Duas derivações diferentes podem levar a uma mesma parse tree

# Parse Trees

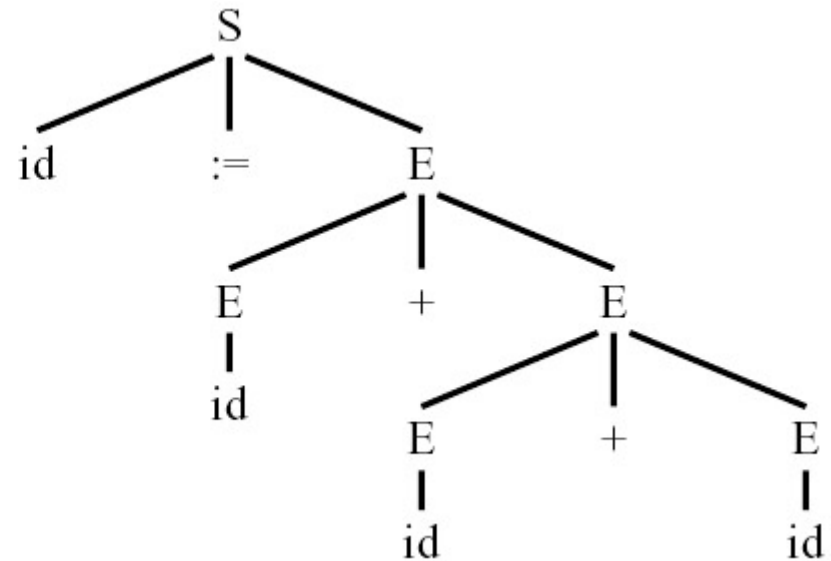
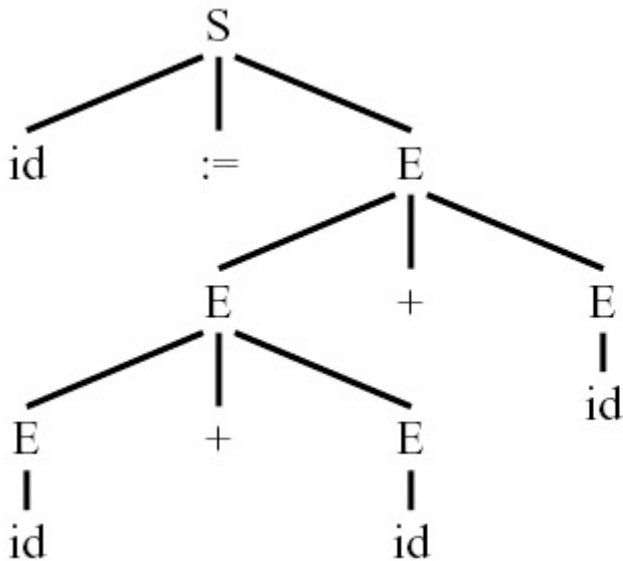
`a := 7; b := c + (d := 5 + 6, d)`



# Gramáticas Ambíguas

- Podem derivar uma sentença com duas *parse trees* diferentes

–  $id := id+id+id$



# É Ambígua?

---

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

- Analise a sentença
  - the girl touches the boy with the flower



# É Ambígua?

---

$E \rightarrow \text{id}$

$E \rightarrow \text{num}$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow E + E$

$E \rightarrow E - E$

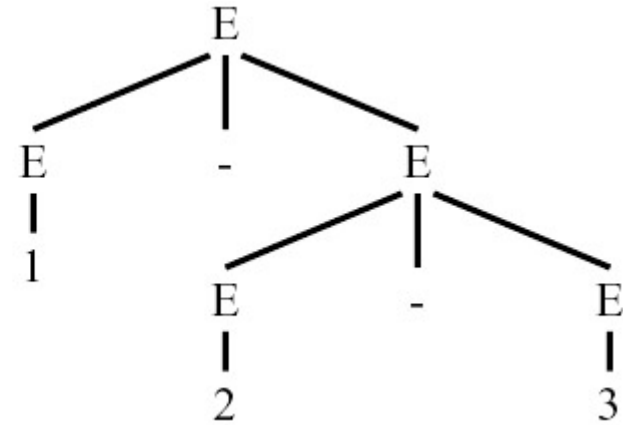
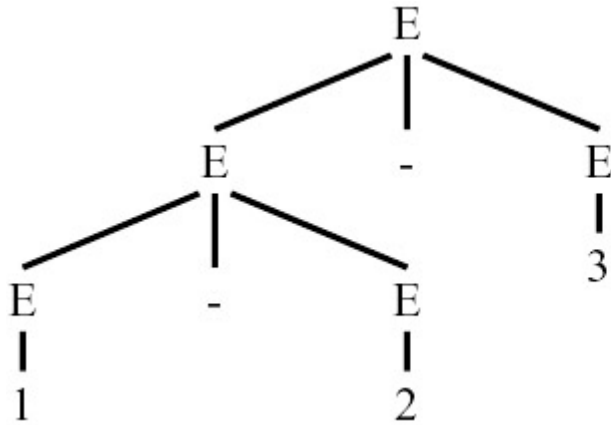
$E \rightarrow (E)$

Construa Parse Trees para as seguintes expressões:

1-2-3

1+2\*3

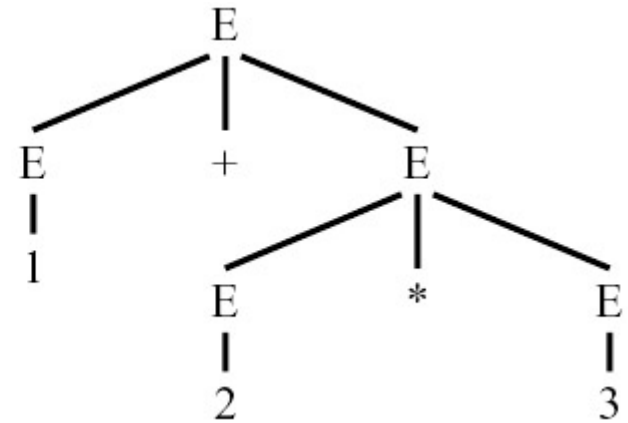
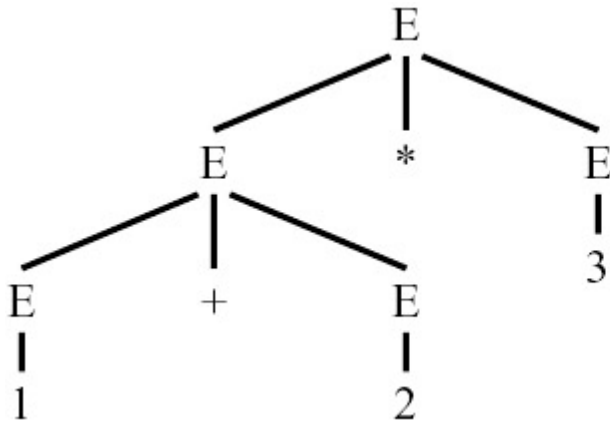
# Exemplo: 1-2-3



**Ambígua!**

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

# Exemplo: $1+2*3$



**Ambígua!**

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

# Gramáticas Ambíguas

---

- Gera uma mesma cadeia com duas árvores sintáticas diferentes
  - E não duas derivações diferentes! Pois podem apresentar a mesma estrutura
- Podemos formalizar assim:
  - Gramáticas ambíguas geram alguma cadeia ambigualmente
  - Uma cadeia é gerada ambigualmente se possui duas ou mais derivações mais à esquerda diferentes.

# Gramáticas Ambíguas

---

- Os compiladores usam as parse trees para extrair o significado das expressões
- A ambiguidade se torna um problema
- Podemos, geralmente, mudar a gramática de maneira a retirar a ambigüidade

# Gramáticas Ambíguas

---

- Alterando o exemplo anterior:
  - Queremos colocar uma precedência maior para \* em relação ou + e –
  - Também queremos que cada operador seja associativo à esquerda:
    - $(1-2)-3$  e não  $1-(2-3)$
- Conseguimos isso introduzindo novos não-terminais

# Gramática 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

# Gramática para Expressões

$$E \rightarrow E + T$$

$$T \rightarrow T * F$$

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

$$E \rightarrow E - T$$

$$T \rightarrow T / F$$

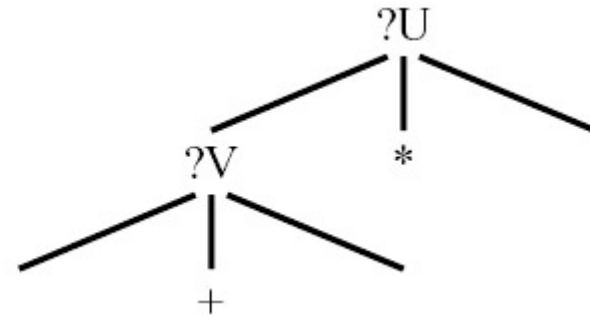
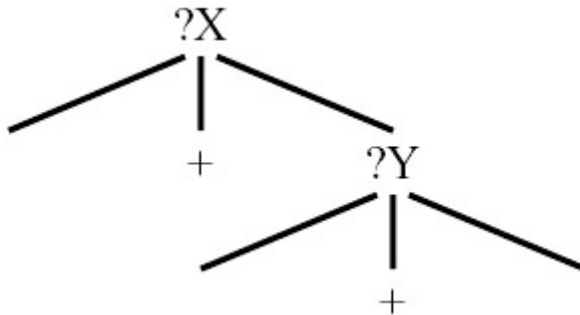
$$F \rightarrow \text{num}$$

$$E \rightarrow T$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

Essa gramática pode gerar as árvores abaixo?





# Gramáticas Ambíguas

---

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

# Fim de Arquivo

---

$$S \rightarrow E \$$$

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow T / F$$

$$T \rightarrow F$$

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

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

$$F \rightarrow (E)$$

Criar um novo não terminal como símbolo inicial

# Parsing

---

- CFG 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)[65,66]
- $O(n^3)$  é muito lento para programas grandes

# Parsing

---

- Existem classes de gramáticas para as quais podemos construir parsers que rodam em tempo linear
- Exemplo:
  - LL: left-to-right, left-most derivation
  - LR: left-to-right, right-most derivation

# Análise Descendente (Predictive Parsing)

---

- Também chamados 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
- Temos uma função para cada não-terminal

# Análise Descendente (Predictive Parsing)

---

$$E \rightarrow +EE$$

$$E \rightarrow *EE$$

$$E \rightarrow a/b$$

- Expressões pré-fixas
- Considere a cadeia  $+b^*ab$
- Como é sua derivação mais à esquerda?

# Análise Descendente (Predictive Parsing)

---

- Análise descendente produz uma derivação à esquerda
- Precisa determinar a produção a ser usada para expandir o não-terminal corrente
- Vejamos um exemplo de implementação

# Análise Descendente

---

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

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

$S \rightarrow \textit{print } E$

$L \rightarrow \textit{end}$

$L \rightarrow ; S L$

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

Como seria um parser  
para essa gramática?



# Análise Descendente

```
final int IF=1, THEN=2, ELSE=3, BEGIN=4, END=5, PRINT=6,
        SEMI=7, NUM=8, EQ=9;
int tok = getToken();
void advance() {tok=getToken();}
void eat(int t) {if (tok==t) advance(); else error();}
void S() {
    switch(tok) {
        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(tok) { case END: eat(END); break;
                 case SEMI: eat(SEMI); S(); L(); break;
                 default: error(); }}
void E() { eat(NUM); eat(EQ); eat(NUM); }
```

# Análise Descendente

---

$$S \rightarrow E \$$$

$$E \rightarrow E + T$$

$$E \rightarrow E - T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow T / F$$

$$T \rightarrow F$$

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

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

$$F \rightarrow (E)$$

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

# Análise Descendente (Predictive Parsing)

---

- Como decidir entre  $E+T$  e  $E$  na função que implementa o não-terminal  $E$ ?
  - Tanto  $E$  como  $T$  podem derivar cadeias começando com  $id$ ,  $num$  ou  $($
  - E se você puder olhar um número  $k > 1$  de símbolos para frente na entrada?

# Análise Descendente (Predictive Parsing)

---

- Como decidir entre  $E+T$  e  $E$  na função que implementa o não-terminal  $E$ ?
  - Tanto  $E$  como  $T$  podem derivar cadeias começando com  $id$ ,  $num$  ou  $($
  - E se você puder olhar um número  $k > 1$  de símbolos para frente na entrada?
    - Essas cadeias podem ter tamanho arbitrário
    - O problema permanece

# Análise Descendente

```
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  $1*2-3$ ?

# FIRST and FOLLOW sets

---

- Dada uma string  $\gamma$  de terminais e não terminais
  - FIRST( $\gamma$ ) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivada de  $\gamma$ .
- Exemplo usando gramática anterior
  - $\gamma = T^*F$ 
    - FIRST( $\gamma$ ) = {id ,num, ( }

# Predictive Parsing

---

- Se uma gramática tem produções da forma:
  - $X \rightarrow \gamma_1$
  - $X \rightarrow \gamma_2$
  - Caso os conjuntos  $FIRST(\gamma_1)$  e  $FIRST(\gamma_2)$  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  $\gamma = X Y Z$  ?*
  - Podemos simplesmente fazer  $\text{FIRST}(XYZ) = \text{FIRST}(X)$  ?



# Resumindo

---

- $\text{Nullable}(X)$  é verdadeiro se  $X$  pode derivar a string vazia
- $\text{FIRST}(\gamma)$  é o conjunto de terminais que podem iniciar strings derivadas de  $\gamma$
- $\text{FOLLOW}(X)$  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  $X Y Z t$ , onde  $Y$  e  $Z$  podem ser vazios

# Definição FIRST, FOLLOW e nullable

---

Os menores conjuntos onde:

for each terminal symbol  $Z$ ,  $\text{FIRST}[Z] = \{Z\}$ .

**for** each production  $X \rightarrow Y_1 Y_2 \cdots Y_k$

**if**  $Y_1 \dots Y_k$  are all nullable (or if  $k = 0$ )

**then**  $\text{nullable}[X] = \text{true}$

**for** each  $i$  from 1 to  $k$ , each  $j$  from  $i + 1$  to  $k$

**if**  $Y_1 \cdots Y_{i-1}$  are all nullable (or if  $i = 1$ )

**then**  $\text{FIRST}[X] = \text{FIRST}[X] \cup \text{FIRST}[Y_i]$

**if**  $Y_{i+1} \cdots Y_k$  are all nullable (or if  $i = k$ )

**then**  $\text{FOLLOW}[Y_i] = \text{FOLLOW}[Y_i] \cup \text{FOLLOW}[X]$

**if**  $Y_{i+1} \cdots Y_{j-1}$  are all nullable (or if  $i + 1 = j$ )

**then**  $\text{FOLLOW}[Y_i] = \text{FOLLOW}[Y_i] \cup \text{FIRST}[Y_j]$

# Algoritmo FIRST, FOLLOW e nullable

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

```
for each terminal symbol  $Z$  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 nullable[ $X$ ]  $\leftarrow$  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 FIRST[ $X$ ]  $\leftarrow$  FIRST[ $X$ ]  $\cup$  FIRST[ $Y_i$ ]
      if  $Y_{i+1} \dots Y_k$  are all nullable (or if  $i = k$ )
        then FOLLOW[ $Y_i$ ]  $\leftarrow$  FOLLOW[ $Y_i$ ]  $\cup$  FOLLOW[ $X$ ]
      if  $Y_{i+1} \dots Y_{j-1}$  are all nullable (or if  $i + 1 = j$ )
        then FOLLOW[ $Y_i$ ]  $\leftarrow$  FOLLOW[ $Y_i$ ]  $\cup$  FIRST[ $Y_j$ ]
    until FIRST, FOLLOW, and nullable did not change in this iteration.
```

# Algoritmo FIRST, FOLLOW e nullable

---

- Algoritmo de iteração até um ponto fixo
- Os conjuntos poderiam ser computados de maneira separada
- Mesmo método usado para  $\epsilon$ -closure
- Aparece também no back-end, para dataflow analysis

# 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	

# Generalizando para strings

---

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

# Construindo um Predictive Parser

---

- Cada função relativa a um não-terminal precisa conter uma cláusula para cada produção
- Precisa saber escolher, baseado no próximo *token*, qual a produção apropriada
- Isto é feito através da *predictive parsing table*



# Construindo um Predictive Parser

---

- Dada uma produção  $X \rightarrow \gamma$
- Para cada  $T \in \text{FIRST}(\gamma)$ 
  - Coloque a produção  $X \rightarrow \gamma$  na linha  $X$ , coluna  $T$ .
- Se  $\gamma$  é *nullable*:
  - Coloque a produção na linha  $X$ , coluna  $T$  para cada  $T \in \text{FOLLOW}[X]$ .

# Exemplo

- $Z \rightarrow d$
- $Z \rightarrow XYZ$
- $Y \rightarrow$
- $Y \rightarrow c$
- $X \rightarrow Y$
- $X \rightarrow a$

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

Funciona ???

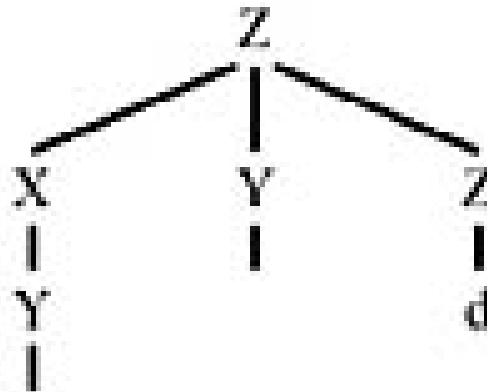
	a	c	d
$X$	$X \rightarrow a$ $X \rightarrow Y$	$X \rightarrow Y$	$X \rightarrow Y$
$Y$	$Y \rightarrow$	$Y \rightarrow$ $Y \rightarrow c$	$Y \rightarrow$
$Z$	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow d$ $Z \rightarrow XYZ$

# Construindo um Predictive Parser

- Não!! 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

---

- 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 uma tabela onde as linhas são os não-terminais e as colunas são todas as seqüências possíveis de  $k$  terminais

# Construindo um Predictive Parser

---

- 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!

# Recursão à Esquerda

---

$S \rightarrow E \$$

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

Consigo gerar um parser  
LL(1) para essa  
gramática?

# Construindo um Predictive Parser

---

- Problema:

- A função que implementa E precisa chamar a si mesma caso escolha E+T.
- Porém, é a primeira ação dela, antes de avançar na cadeia de entrada
- Laço infinito!
- Acontece devido à recursão à esquerda
  - $E \rightarrow E + T$
  - $E \rightarrow E - T$
  - $T \rightarrow T * F$

# Recursão à Esquerda

$$S \rightarrow E \$$$

$$E \rightarrow E - T$$

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow T / F$$

$$T \rightarrow F$$

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

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

$$F \rightarrow (E)$$

Gramáticas com recursão à esquerda não podem ser LL(1)

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

Fatoração (recursão à direita)!



# Recursão à Esquerda

Generalizando:

- Tendo  $X \rightarrow X\gamma$  e  $X \rightarrow \alpha$ , onde  $\alpha$  não começa com  $X$
- Derivamos strings da forma  $\alpha\gamma^*$ 
  - $\alpha$  seguido de zero ou mais  $\gamma$ .
- Podemos reescrever:

$$\begin{pmatrix} X \rightarrow X\gamma_1 \\ X \rightarrow X\gamma_2 \\ X \rightarrow \alpha_1 \\ X \rightarrow \alpha_2 \end{pmatrix} \Rightarrow \begin{pmatrix} X \rightarrow \alpha_1 X' \\ X \rightarrow \alpha_2 X' \\ X' \rightarrow \gamma_1 X' \\ X' \rightarrow \gamma_2 X' \\ X' \rightarrow \end{pmatrix}$$

# Eliminando Recursão à Esquerda

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

	nullable	FIRST	FOLLOW
$S$	no	( id num	
$E$	no	( id num	) \$
$E'$	yes	+ -	) \$
$T$	no	( id num	) + - \$
$T'$	yes	* /	) + - \$
$F$	no	( id num	) * / + - \$

# Eliminando Recursão à Esquerda

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

	+	*	id	(	)	\$
$S$			$S \rightarrow ES$	$S \rightarrow ES$		
$E$			$E \rightarrow TE'$	$E \rightarrow TE'$		
$E'$	$E' \rightarrow +TE'$				$E' \rightarrow$	$E' \rightarrow$
$T$			$T \rightarrow FT'$	$T \rightarrow FT'$		
$T'$	$T' \rightarrow$	$T' \rightarrow *FT'$			$T' \rightarrow$	$T' \rightarrow$
$F$			$F \rightarrow \text{id}$	$F \rightarrow (E)$		

# Fatoração à Esquerda

---

- Um outro problema para predictive parsing ocorre em situações do tipo:

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

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

- Regras do mesmo não terminal começam com os mesmo símbolos

# Fatoração à Esquerda

---

- Criar um novo não-terminal para os finais permitidos:

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

- Gramática ainda é ambígua, mas conflito pode ser resolvido escolhendo sempre a regra para else.

# Recuperação de Erros

---

- Uma entrada em branco na tabela indica um caractere não esperado
- Parar o processo no primeiro erro encontrado não é desejável
- Duas alternativas:
  - Inserir símbolo:
    - Assume que encontrou o que esperava
  - Deletar símbolo(s):
    - Pula tokens até que um elemento do FOLLOW seja atingido.

# Recuperação de Erros

---

```
void T() {
    switch (tok) {
        case ID:
        case NUM:
        case LPAREN: F(); Tprime(); break;
        default: print("expected id, num, or
left-paren");
    }
}
```

# Recuperação de Erros

---

```
int Tprime_follow [] = {PLUS, RPAREN, EOF};
void Tprime() {
    switch (tok) {
        case PLUS: break;
        case TIMES: eat(TIMES); F(); Tprime(); break;
        case RPAREN: break;
        case EOF: break;
        default: print("expected +, *, right-paren, or
end-of-file");
        skipto(Tprime_follow);
    }
}
```