Análise Sintática

Universidade Federal do Ceará - Campus de Quixadá

Lucas Ismaily
ismailybf@ufc.br

Semestre 2022.2

Linguagens de Programação
Baseado nos slides do Prof. Sandro Rigo (IC-Unicamp)

Seção 1

Introdução

Front-end

- Análise Léxica:

Front-end

- Análise Léxica:
  - Quebra a entrada em palavras conhecidas como símbolos léxicos (tokens).

Front-end

- Análise Léxica:
  - Quebra a entrada em palavras conhecidas como símbolos léxicos (tokens).
- Análise Sintática:

Front-end

- Análise Léxica:
  - Quebra a entrada em palavras conhecidas como símbolos léxicos (tokens).
- Análise Sintática:
  - Analisa a estrutura de frases do programa.

Front-end

- Análise Léxica:
  - Quebra a entrada em palavras conhecidas como símbolos léxicos (tokens).
- Análise Sintática:
  - Analisa a estrutura de frases do programa.
- Análise Semântica:

Front-end

- Análise Léxica:
  - Quebra a entrada em palavras conhecidas como símbolos léxicos (tokens).
- Análise Sintática:
  - Analisa a estrutura de frases do programa.
- Análise Semântica:
  - Calcula o significado do programa.

Analisador Sintático (Parser)

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

Analisador Sintático (Parser)

- Recebe uma sequência 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.

Analisador Sintático (Parser)

- Recebe uma sequência 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.

Gramáticas livres de contexto

- ERs são boas para definir a estrutura léxica de maneira declarativa.

Gramáticas livres de contexto

- ERs são boas para definir a estrutura léxica de maneira declarativa.
- Entretanto, não são "poderosas" o suficiente para definir declarativamente a estrutura sintática de linguagens de programação.

Gramáticas livres de contexto

- Exemplo de ER usando abreviações:

Gramáticas livres de contexto

- Exemplo de ER usando abreviações:
  - digits $= [0-9]^+$

Gramáticas livres de contexto

- Exemplo de ER usando abreviações:
  - digits $= [0-9]^+$
  - sum $=$ (digits "$+$")* digits

Gramáticas livres de contexto

- Exemplo de ER usando abreviações:
  - digits $= [0-9]^+$
  - sum $=$ (digits "+")* digits
  - definem somas da forma 28+301+9

Gramáticas livres de contexto

- Exemplo de ER usando abreviações:
  - digits $= [0-9]^+$
  - sum $=$ (digits "+")* digits
  - definem somas da forma 28+301+9

- Como isso é implementado?

Gramáticas livres de contexto

- 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.

Gramáticas livres de contexto

- 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 $= ([0-9]^+$ "+")$^*[0-9]^+$

Gramáticas livres de contexto

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

Gramáticas livres de contexto

- É possível usar a mesma ideia para definir uma linguagem para expressões que tenham parênteses balanceados?
  - (1+(245+2))

Gramáticas livres de contexto

- É possível usar a mesma ideia para definir uma linguagem para expressões que tenham parênteses balanceados?
  - (1+(245+2))

- Tentativa

Gramáticas livres de contexto

- É possível usar a mesma ideia para definir uma linguagem para expressões que tenham parênteses balanceados?
  - (1+(245+2))

- Tentativa
  - digits $= [0-9]^+$

Gramáticas livres de contexto

- É possível usar a mesma ideia para definir uma linguagem para expressões que tenham parênteses balanceados?
  - (1+(245+2))

- Tentativa
  - digits $= [0-9]^+$
  - sum $=$ expr "$+$" expr

Gramáticas livres de contexto

- É possível usar a mesma ideia para definir uma linguagem para expressões que tenham parênteses balanceados?
  - (1+(245+2))

- Tentativa
  - digits $= [0-9]^+$
  - sum $=$ expr "$+$" expr
  - expr $=$ "(" sum ")" | digits

Gramáticas livres de contexto

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

Gramáticas livres de contexto

- O analisador léxico substituiria *sum* em *expr*:
  - expr = “(” expr “+” expr “)” | digits

Gramáticas livres de contexto

- O analisador léxico substituiria *sum* em *expr*:
  - expr = “(” expr “+” expr “)” | digits
- Depois substituiria *expr* no próprio *expr*:

Gramáticas livres de contexto

- O analisador léxico substituiria *sum* em *expr*:
  - expr = “(” expr “+” expr “)” | digits
- Depois substituiria *expr* no próprio *expr*:
  - expr = “(”(“(” expr “+” expr “)” | digits) “+” expr “)” | digits

Gramáticas livres de contexto

- 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!

Gramáticas livre de contexto

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

Gramáticas livre de contexto

- As abreviações não acrescentam o poder de expressar recursão às ERs.
- É isso que precisamos para expressar a recursão mútua entre *sum* e *expr*.

Gramáticas livre de contexto

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

Gramáticas livre de contexto

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

```
expr = ab(c|d)e  ⟹  aux = c|d
                     expr = a b aux e
```

Gramáticas livre de contexto

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

Gramáticas livre de contexto

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

$$\texttt{symbol} \longrightarrow \texttt{symbol symbol symbol ... symbol}$$

Gramáticas livre de contexto

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

```
symbol ⟶ symbol symbol symbol ... symbol
```

onde existem zero ou mais símbolos do lado direito.

Gramáticas livre de contexto

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

```
symbol ⟶ symbol symbol symbol ... symbol
```

onde existem zero ou mais símbolos do lado direito.

Símbolos:

Gramáticas livre de contexto

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

```
symbol ⟶ symbol symbol symbol ... symbol
```

onde existem zero ou mais símbolos do lado direito.

Símbolos:

- terminais: uma string do alfabeto da linguagem.

Gramáticas livre de contexto

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

$$\texttt{symbol} \longrightarrow \texttt{symbol symbol symbol ... symbol}$$

onde existem zero ou mais símbolos do lado direito.

Símbolos:

- terminais: uma string do alfabeto da linguagem.
- não-terminais: aparecem do lado esquerdo de alguma produção.

Gramáticas livre de contexto

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

```
symbol ⟶ symbol symbol symbol ... symbol
```

onde existem zero ou mais símbolos do lado direito.

Símbolos:

- terminais: uma string do alfabeto da linguagem.
- não-terminais: aparecem do lado esquerdo de alguma produção.
- nenhum token aparece do lado esquedo de uma produção.

Gramáticas livre de contexto

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

```
symbol ⟶ symbol symbol symbol ... symbol
```

onde existem zero ou mais símbolos do lado direito.

Símbolos:

- terminais: uma string do alfabeto da linguagem.
- não-terminais: aparecem do lado esquerdo de alguma produção.
- nenhum token aparece do lado esquedo de uma produção.
- existem um não-terminal definido como *start symbol*.

Gramáticas livre de contexto

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

Gramáticas livre de contexto

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

Gerar cadeias da linguagem:

Gramáticas livre de contexto

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

Gerar cadeias da linguagem:

1. Escreva a variável inicial.

Gramáticas livre de contexto

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

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.

Gramáticas livre de contexto

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

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.

Gramáticas livre de contexto

- A sequência de substituições é chamada de derivação.
- Ex.:

Gramáticas livre de contexto

- A sequência de substituições é chamada de derivação.
- Ex.:
  - 000#111

Gramáticas livre de contexto

- A sequência de substituições é chamada de derivação.
- Ex.:
  - 000#111
  - $A \longrightarrow 0A1 \longrightarrow 00A11 \ldots$

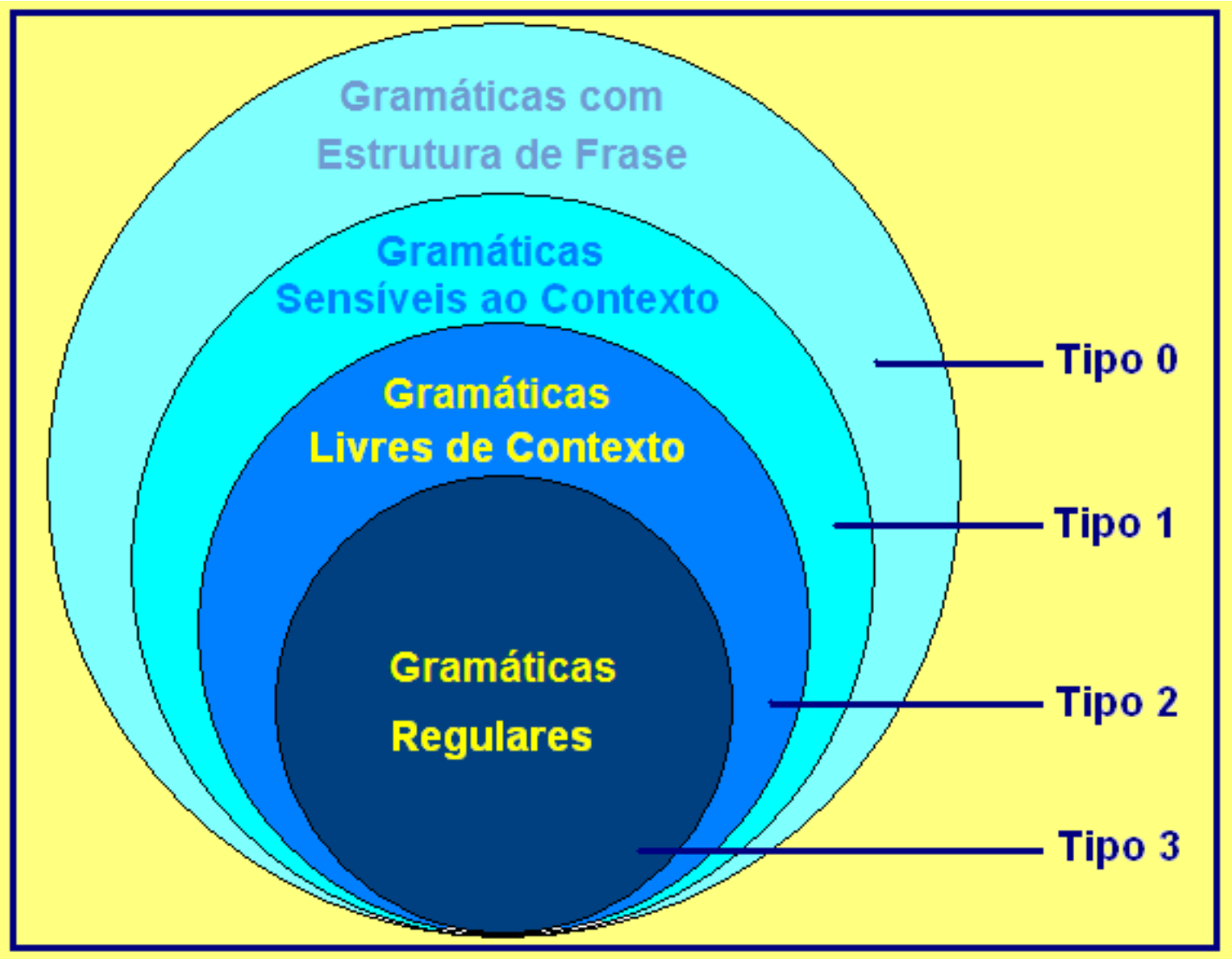
Gramáticas livre de contexto

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

Hierarquia de Chomsky

Teoria de autômatos: linguagem formal e gramática formal			
Hierarquia Chomsky	**Gramática**	**Linguagem**	**Reconhecedor**
Tipo-0	Irrestrita	Recursivamente enumerável	Máquina de Turing
--	--	Recursiva	Máquina de Turing que sempre para
Tipo-1	Sensível ao contexto	Sensível ao contexto	Autômato linearmente limitado
Tipo-2	Livre de contexto	Livre de contexto	Autômato com pilha
Tipo-3	Regular	Regular	Autômato finito

Hierarquia de Chomsky



Gramáticas livre de contexto

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

Gramáticas livre de contexto

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

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

Gramáticas livre de contexto

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

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

Possível código fonte:

Gramáticas livre de contexto

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

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

Possível código fonte:

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

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

$$\underline{S}$$

$$S\ ;\ \underline{S}$$

$$\underline{S}\ ;\ \text{id} := E$$

$$\text{id} := \underline{E}\ ;\ \text{id} := E$$

$$\text{id} := \text{num}\ ;\ \text{id} := \underline{E}$$

$$\text{id} := \text{num}\ ;\ \text{id} := E + \underline{E}$$

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

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

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

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

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

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

$$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \text{num} + \text{num}\ ,\ \text{id}\ )$$

Derivações

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

Derivações

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

Derivações

- *left-most:* o não terminal mais a esquerda é sempre o expandido;
- *right-most:* o não terminal mais a direita é sempre o expandido.
- Qual é o caso do exemplo anterior?

Parse Trees

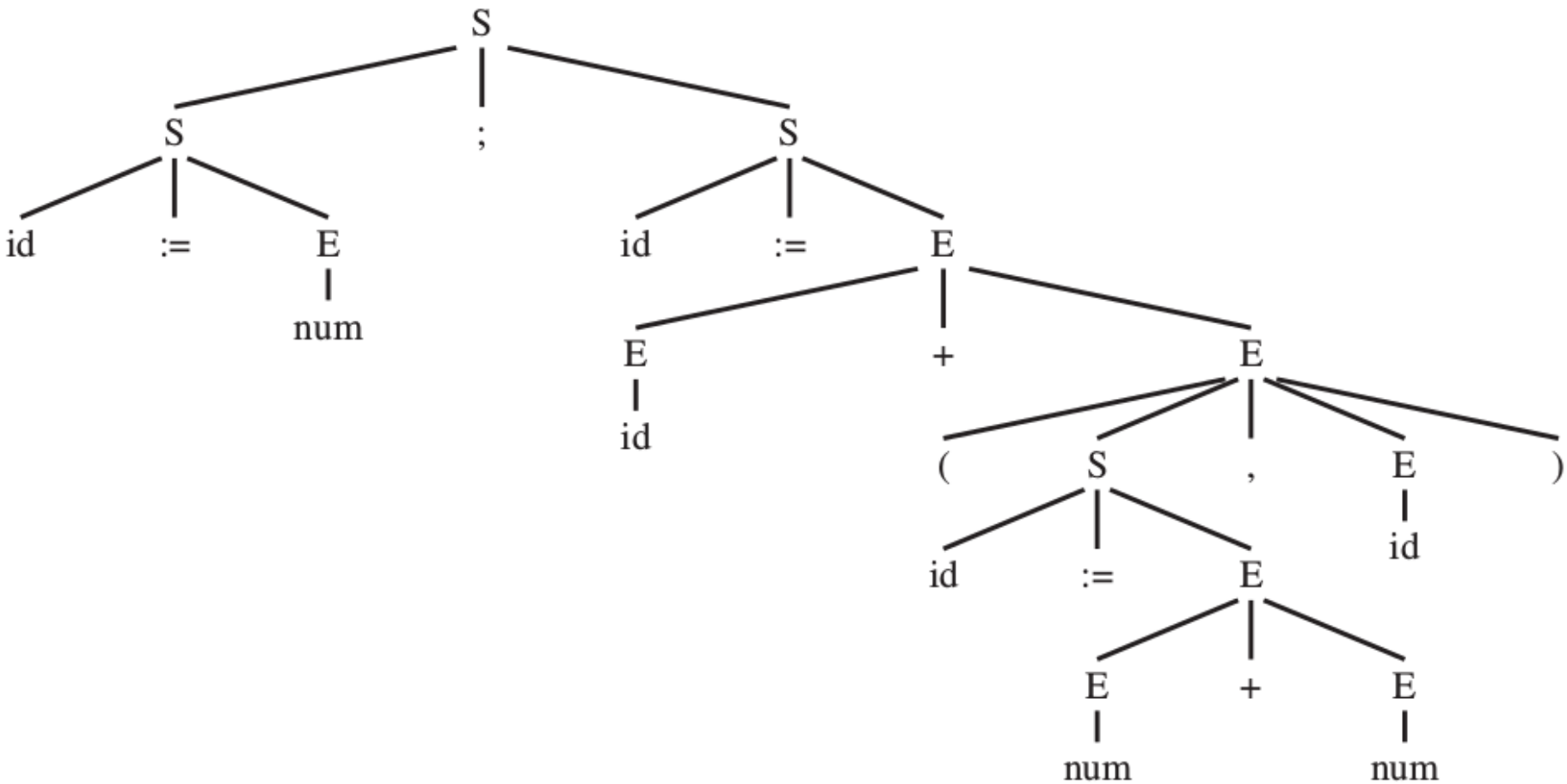
- Constrói-se uma árvore conectando-se cada símbolo em uma derivação ao qual ele foi derivado.

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 Tree

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

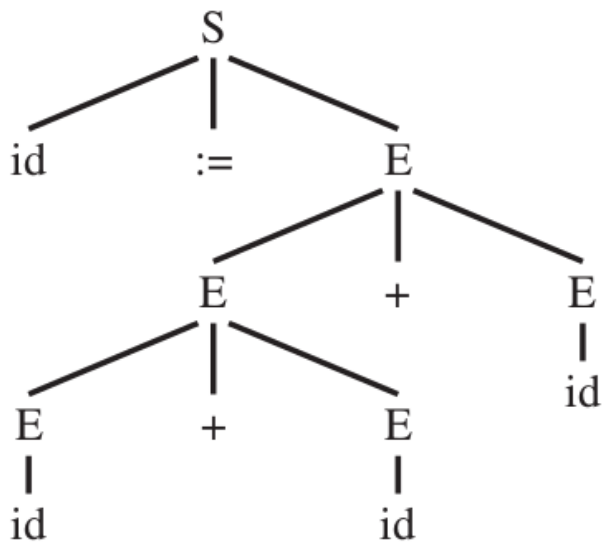


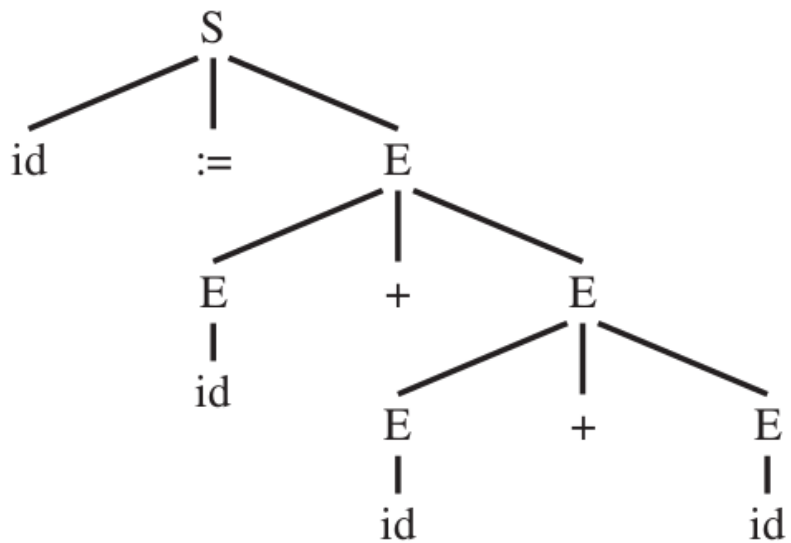
Gramáticas Ambíguas

- Podem derivar uma sentença com duas *parse trees* diferentes.
  - `id := id + id + id`

Gramáticas Ambíguas

- Podem derivar uma sentença com duas *parse trees* diferentes.
  - `id := id + id + id`





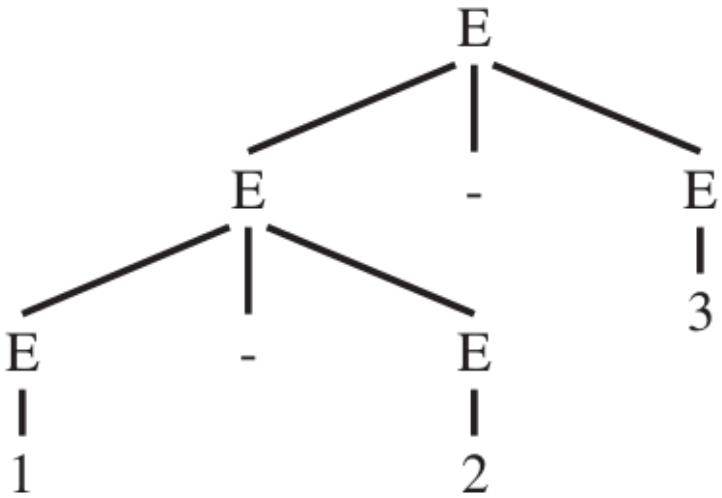
É Ambígua???

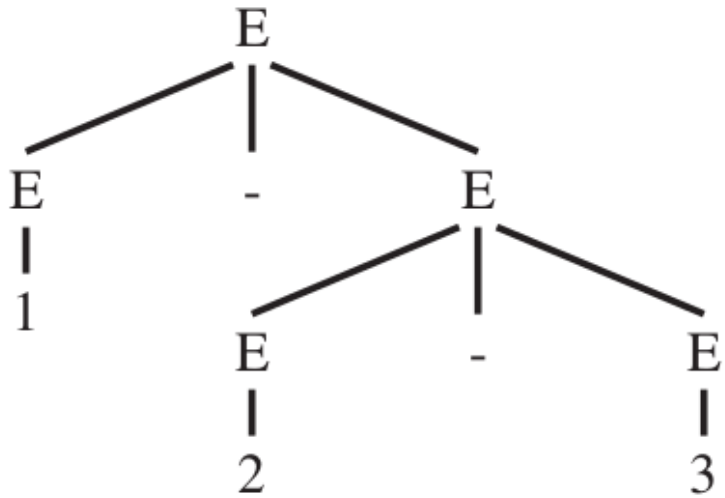
1. E $\longrightarrow$ id
2. E $\longrightarrow$ num
3. E $\longrightarrow$ E * E
4. E $\longrightarrow$ E/E
5. E $\longrightarrow$ E + E
6. E $\longrightarrow$ E - E
7. E $\longrightarrow$ (E)

Construa *Parse Trees* para as seguintes expressões:

- 1-2-3
- 1+2*3

Exemplo 1-2-3

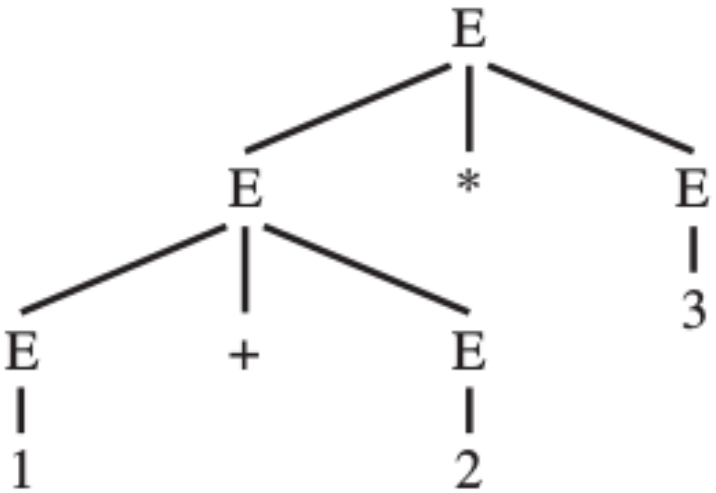


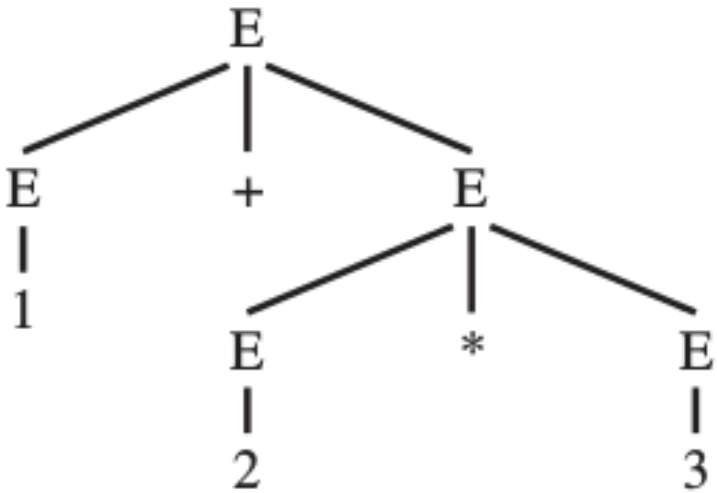


Ambígua!!!

(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

- Os compiladores usam as *parse trees* para extrair o significado das expressões.

Gramáticas Ambíguas

- Os compiladores usam as *parse trees* para extrair o significado das expressões.
- A ambiguidade se torna um problema.

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 ambiguidade.

Gramáticas Ambíguas

- Alterando o exemplo anterior:

Gramáticas Ambíguas

- Alterando o exemplo anterior:
  - Queremos colocar uma precedência maior para * em relação ao + e ao -.

Gramáticas Ambíguas

- Alterando o exemplo anterior:
  - Queremos colocar uma precedência maior para * em relação ao + e ao -.
  - Também queremos que cada operador seja associado à esquerda:

Gramáticas Ambíguas

- Alterando o exemplo anterior:
  - Queremos colocar uma precedência maior para * em relação ao + e ao -.
  - Também queremos que cada operador seja associado à esquerda:
    - (1-2)-3 e não 1-(2-3)

Gramáticas Ambíguas

- Alterando o exemplo anterior:
  - Queremos colocar uma precedência maior para * em relação ao + e ao -.
  - Também queremos que cada operador seja associado à esquerda:
    - (1-2)-3 e não 1-(2-3)
- Conseguimos introduzindo novos não-terminais.

Gramática para Expressões

1. $E \longrightarrow E + T$
2. $E \longrightarrow E$ - T
3. $E \longrightarrow T$
4. $T \longrightarrow T$ * F
5. $T \longrightarrow T/F$
6. $T \longrightarrow F$
7. $F \longrightarrow$ id
8. $F \longrightarrow$ num
9. $F \longrightarrow (E)$

Gramática para Expressões

1. $E \longrightarrow E + T$
2. $E \longrightarrow E$ - T
3. $E \longrightarrow T$
4. $T \longrightarrow T$ * F
5. $T \longrightarrow T/F$
6. $T \longrightarrow F$
7. $F \longrightarrow$ id
8. $F \longrightarrow$ num
9. $F \longrightarrow (E)$

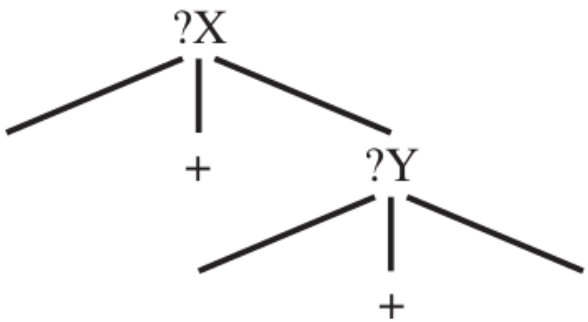
Construa *Parse Trees* para as seguintes expressões:

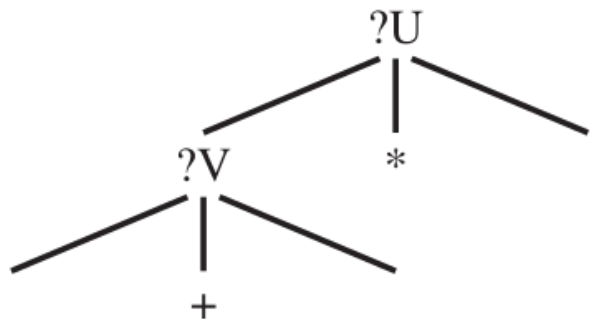
- 1-2-3
- 1+2*3

Gramática para Expressões

1. $E \longrightarrow E + T$
2. $E \longrightarrow E - T$
3. $E \longrightarrow T$
4. $T \longrightarrow T * F$
5. $T \longrightarrow T/F$
6. $T \longrightarrow F$
7. $F \longrightarrow id$
8. $F \longrightarrow num$
9. $F \longrightarrow (E)$

Essa gramática pode gerar a árvore abaixo?





Gramáticas Ambíguas

- Geralmente podemos transformar uma gramática para retirar a ambiguidade.

Gramáticas Ambíguas

- Geralmente podemos transformar uma gramática para retirar a ambiguidade.
- Algumas linguagens não possuem gramáticas não ambíguas.

Gramáticas Ambíguas

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

Fim de Arquivo

0. $S \longrightarrow E\ \$$
1. $E \longrightarrow E + T$
2. $E \longrightarrow E - T$
3. $E \longrightarrow T$
4. $T \longrightarrow T * F$
5. $T \longrightarrow T/F$
6. $T \longrightarrow F$
7. $F \longrightarrow id$
8. $F \longrightarrow num$
9. $F \longrightarrow (E)$

Criar um novo não terminal como símbolo inicial.

Parsing

- CFG (*context free grammar*) geram as linguagens.

Parsing

- CFG (*context free grammar*) geram as linguagens.
- Parsers são reconhecedores das linguagens.

Parsing

- CFG (*context free grammar*) geram as linguagens.
- Parsers são reconhecedores das linguagens.
- Para qualquer CFG é possível obter um parser que roda em O(n^3).

Parsing

- CFG (*context free grammar*) 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 e CYK(Cocke-Younger-Kasami).

Parsing

- CFG (*context free grammar*) 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 e CYK(Cocke-Younger-Kasami).
- 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.

Parsing

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

Parsing

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

Análise Descendente (Predictive Parsing)

- Também chamados de *recursive-descent* ou *top-down*.
- É um algoritmo simples, capaz de fazer o *parsing* de algumas gramáticas (gramáticas LL).

Análise Descendente (Predictive Parsing)

- Também chamados de *recursive-descent* ou *top-down*.
- É um algoritmo simples, capaz de fazer o *parsing* de algumas gramáticas (gramáticas LL).
- Cada produção se torna uma cláusula em uma função recursiva.

Análise Descendente (Predictive Parsing)

- Também chamados de *recursive-descent* ou *top-down*.
- É um algoritmo simples, capaz de fazer o *parsing* de algumas gramáticas (gramáticas LL).
- Cada produção se torna uma cláusula em uma função recursiva.
- Temos uma função para cada não-terminal.

Análise Descendente (Predictive Parsing)

$$
\begin{array}{lcl}
E & \longrightarrow & +EE \\
E & \longrightarrow & *EE \\
E & \longrightarrow & a|b
\end{array}
$$

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

Análise Descendente (Predictive Parsing)

- Análise dessedente produz uma derivação à esquerda.

Análise Descendente (Predictive Parsing)

- Análise dessedente produz uma derivação à esquerda.
- Precisa determinar a produção a ser usada para expandir o não-terminal corrente.

Análise Descendente (Predictive Parsing)

- Análise dessedente 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 (Predictive Parsing)

$$
\begin{array}{lcl}
S & \longrightarrow & \texttt{if E then S else S} \\
S & \longrightarrow & \texttt{begin S L} \\
S & \longrightarrow & \texttt{print E} \\
L & \longrightarrow & \texttt{end} \\
L & \longrightarrow & \texttt{; S L} \\
E & \longrightarrow & \texttt{num = num}
\end{array}
$$

Como seria um parser para essa gramática?

Análise Descendente (Predictive Parsing)

```
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 (Predictive Parsing)

0. $S \longrightarrow E\ \$$
1. $E \longrightarrow E + T$
2. $E \longrightarrow E - T$
3. $E \longrightarrow T$
4. $T \longrightarrow T * F$
5. $T \longrightarrow T/F$
6. $T \longrightarrow F$
7. $F \longrightarrow id$
8. $F \longrightarrow num$
9. $F \longrightarrow (E)$

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

Análise Descendente (Predictive Parsing)

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

Análise Descendente (Predictive Parsing)

- Como decidir entre E+T e T na função que implementa o não-terminal E?
  - Tanto E como T podem derivar cadeias começando com id ou "(".

Análise Descendente (Predictive Parsing)

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

Análise Descendente (Predictive Parsing)

- Como decidir entre E+T e T na função que implementa o não-terminal E?
  - Tanto E como T podem derivar cadeias começando com id ou "(".
  - E se você puder olhar o número k>1 para frente da entrada?
- Essas cadeias podem ter tamanho arbitrário.

Análise Descendente (Predictive Parsing)

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

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??

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?

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?

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais
  - FIRST(y) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivadas de y.

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais
  - FIRST(y) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivadas de y.
  - FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais
  - FIRST(y) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivadas de y.
  - FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.
  - $t \in$ FOLLOW(X) se existe alguma derivação contendo Xt.

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais
  - FIRST(y) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivadas de y.
  - FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.
  - $t \in$ FOLLOW(X) se existe alguma derivação contendo Xt.
  - Cuidado com derivações da forma XYZt, onde Y e Z podem ser vazios.

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais
  - FIRST(y) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivadas de y.
  - FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.
  - $t \in$ FOLLOW(X) se existe alguma derivação contendo Xt.
  - Cuidado com derivações da forma XYZt, onde Y e Z podem ser vazios.
- Exemplo usando a gramática anterior:

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais
  - FIRST(y) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivadas de y.
  - FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.
  - $t \in$ FOLLOW(X) se existe alguma derivação contendo Xt.
  - Cuidado com derivações da forma XYZt, onde Y e Z podem ser vazios.
- Exemplo usando a gramática anterior:
  - $y =$ T*F

Conjuntos FIRST e FOLLOW

- Dada uma string y de terminais e não terminais
  - FIRST(y) é o conjunto de todos os terminais que podem iniciar uma string de terminais derivadas de y.
  - FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.
  - $t \in$ FOLLOW(X) se existe alguma derivação contendo Xt.
  - Cuidado com derivações da forma XYZt, onde Y e Z podem ser vazios.
- Exemplo usando a gramática anterior:
  - $y =$ T*F
    - FIRST(y) $=$ {id, num, (}

Algoritmo para calcular os conjuntos FIRST e FOLLOW

for each terminal symbol Z
    FIRST[Z] $\leftarrow$ {Z}
repeat
    for each production $X \rightarrow Y_1 Y_2 \cdots Y_k$
        if $Y_1 \ldots Y_k$ 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 $Y_1 \cdots Y_{i-1}$ are all nullable (or if $i = 1$)
              then FIRST[X] $\leftarrow$ FIRST[X] $\cup$ FIRST[Y_i]
            if $Y_{i+1} \cdots Y_k$ are all nullable (or if $i = k$)
              then FOLLOW[Y_i] $\leftarrow$ FOLLOW[Y_i] $\cup$ FOLLOW[X]
            if $Y_{i+1} \cdots Y_{j-1}$ 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.

Análise Descendente (Predictive Parsing)

- Se uma gramática tem produções da forma:

Análise Descendente (Predictive Parsing)

- Se uma gramática tem produções da forma:
  - X $\longrightarrow y1$

Análise Descendente (Predictive Parsing)

- Se uma gramática tem produções da forma:
  - $X \longrightarrow y1$
  - $X \longrightarrow y2$

Análise Descendente (Predictive Parsing)

- Se uma gramática tem produções da forma:
  - X $\longrightarrow y1$
  - X $\longrightarrow y2$
    - Caso os conjuntos FIRST($y1$) e FIRST($y2$) tenham intersecção, então a gramática não pode ser analisada com um predictive parser.

Análise Descendente (Predictive Parsing)

- Se uma gramática tem produções da forma:
  - X $\longrightarrow y1$
  - X $\longrightarrow y2$
    - Caso os conjuntos FIRST($y1$) e FIRST($y2$) tenham intersecção, então a gramática não pode ser analisada com um predictive parser.
- Por que?

Análise Descendente (Predictive Parsing)

- Se uma gramática tem produções da forma:
  - X $\longrightarrow y1$
  - X $\longrightarrow y2$
    - Caso os conjuntos FIRST($y1$) e FIRST($y2$) 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

$$
\begin{array}{lcl}
Z & \longrightarrow & d \\
Z & \longrightarrow & XYZ \\
Y & \longrightarrow & \\
Y & \longrightarrow & c \\
X & \longrightarrow & Y \\
X & \longrightarrow & a
\end{array}
$$

- Como seria para $y=$ XYZ?

Calculando FIRST

$$
\begin{array}{lcl}
Z & \longrightarrow & d \\
Z & \longrightarrow & XYZ \\
Y & \longrightarrow & \\
Y & \longrightarrow & c \\
X & \longrightarrow & Y \\
X & \longrightarrow & a
\end{array}
$$

- Como seria para $y=$ XYZ?
- Podemos simplesmente fazer FIRST(XYZ) $=$ FIRST(X)?

Resumindo

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

Resumindo

- Nullable(X) é verdadeiro se X pode derivar a string vazia.
- FIRST(y) é o conjunto de terminais que podem iniciar strings derivadas de y.

Resumindo

- Nullable(X) é verdadeiro se X pode derivar a string vazia.
- FIRST(y) é o conjunto de terminais que podem iniciar strings derivadas de y.
- FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.

Resumindo

- Nullable(X) é verdadeiro se X pode derivar a string vazia.
- FIRST(y) é o conjunto de terminais que podem iniciar strings derivadas de y.
- FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X.
  - $t \in$ FOLLOW(X) se existe alguma derivação contendo Xt.

Resumindo

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

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 tabela do predictive parsing.

Construindo um Predictive Parser

- Dada uma produção X $\longrightarrow \gamma$.
- Para cada T $\in$ FIRST(γ)
  - Coloque a produção X $\longrightarrow \gamma$ na linha X, coluna T.
- Se γ é nullable
  - Coloque a produção na linha X, coluna T para cada T $\in$ FOLLOW[X].

Exemplo

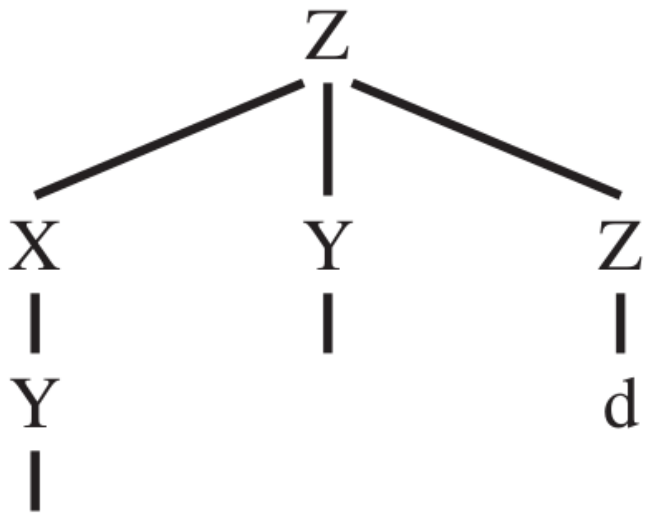
$$
\begin{aligned}
Z &\longrightarrow d \\
Z &\longrightarrow XYZ \\
Y &\longrightarrow \\
Y &\longrightarrow c \\
X &\longrightarrow Y \\
X &\longrightarrow a
\end{aligned}
$$

	nullable	FIRST	FOLLOW
X	true	a,c	a,c,d
Y	true	c	a,c,d
Z	false	a,c,d	-

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

Construindo um Predictive Parser

- Não é possível fazer um parser predictive para essa gramática porque ela é ambígua.
  - Note que algumas células da tabela do predictive parser têm mais de uma entrada!
  - Isso sempre acontece com gramáticas ambíguas (Mas pode acontece também em gramáticas não ambíguas.)



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 sequências possíveis de k terminais.

Construindo um Predictive Parser

- Isso raramente é feito devido ao tamanho explosivo das tabelas geradas (deve ter um simbolo para cada combinação, não-terminal-lookahead)
- Gramáricas analisáveis com tabelas LL(K) são chamadas LL(K).
- Nenhuma gramática ambígua é LL(K) para nenhum k!

Recursão à Esquerda

0. $S \longrightarrow E\ \$$
1. $E \longrightarrow E + T$
2. $E \longrightarrow E - T$
3. $E \longrightarrow T$
4. $T \longrightarrow T * F$
5. $T \longrightarrow T/F$
6. $T \longrightarrow F$
7. $F \longrightarrow id$
8. $F \longrightarrow num$
9. $F \longrightarrow (E)$

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

Recursão à esquerda

- 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.
- Como Resolver??? (Fatoração - recursão à direita).

$$
\begin{aligned}
E &\longrightarrow E+T \\
E &\longrightarrow E\text{-}T \\
T &\longrightarrow T*F
\end{aligned}
\qquad\qquad
\begin{aligned}
E &\longrightarrow TE' \\
E' &\longrightarrow +TE' \\
E' &\longrightarrow
\end{aligned}
$$

Recursão à esquerda

- Generalizando:
  - Tendo X $\longrightarrow$ Xγ e X $\longrightarrow \alpha$, onde α não começa com X.
- Derivamos strings da forma $\alpha\gamma^*$
  - α seguindo de zero ou mais γ.
- Podemos reescrever:

$$\begin{pmatrix} X \to X\ \gamma_1 \\ X \to X\ \gamma_2 \\ X \to \alpha_1 \\ X \to \alpha_2 \end{pmatrix} \Longrightarrow \begin{pmatrix} X \to \alpha_1\ X' \\ X \to \alpha_2\ X' \\ X' \to \gamma_1\ X' \\ X' \to \gamma_2\ X' \\ X' \to \end{pmatrix}$$

Eliminando Recursão à Esquerda

0. S $\longrightarrow$ E $
1. E $\longrightarrow$ TE'
2. E $\longrightarrow$ +TE'
3. E' $\longrightarrow$ -TE'
4. E' $\longrightarrow$
5. T $\longrightarrow$ FT'
6. T' $\longrightarrow$ *FT'
7. T' $\longrightarrow$ /FT'
8. T' $\longrightarrow$
9. F $\longrightarrow$ id
10. F $\longrightarrow$ num
11. F $\longrightarrow$ (E)

	nullable	FIRST	FOLLOW
S	false	(id num	
E	false	(id num	)$
E'	True	+ -	)$
T	false	(id num	)$ +-
T'	True	* /	)$ +-
F	false	(id num	)*/+-$

Eliminando Recursão à Esquerda

0. S $\longrightarrow$ E $
1. E $\longrightarrow$ TE'
2. E $\longrightarrow$ +TE'
3. E' $\longrightarrow$ -TE'
4. E' $\longrightarrow$
5. T $\longrightarrow$ FT'
6. T' $\longrightarrow$ *FT'
7. T' $\longrightarrow$ /FT'
8. T' $\longrightarrow$
9. F $\longrightarrow$ id
10. F $\longrightarrow$ num
11. F $\longrightarrow$ (E)

	+	*	id	(	)	$
S			S $\longrightarrow$ E$	S $\longrightarrow$ E$		
E			E $\longrightarrow$ TE'	E $\longrightarrow$ TE'		
E'	E' $\longrightarrow$ +TE'				E' $\longrightarrow$	E' $\longrightarrow$
T			T $\longrightarrow$ FT'	T $\longrightarrow$ FT'		
T'	T' $\longrightarrow$	T' $\longrightarrow$ *FT'			T' $\longrightarrow$	T' $\longrightarrow$
F			F $\longrightarrow$ id	F $\longrightarrow$ (E)		

Fatoração à esquerda

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

$$\begin{aligned} S &\longrightarrow \texttt{if E then S else S} \\ S &\longrightarrow \texttt{if E then S} \end{aligned}$$

- Regras do mesmo não terminal começam com os mesmos símbolos.

Fatoração à esquerda

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

$$
\begin{aligned}
S &\longrightarrow \texttt{if E then S X} \\
X &\longrightarrow \\
X &\longrightarrow \texttt{else S}
\end{aligned}
$$

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

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);
      }}
```

Análise Sintática

Universidade Federal do Ceará - Campus de Quixadá

Lucas Ismaily
ismailybf@ufc.br

Semestre 2022.2

Linguagens de Programação
Baseado nos slides do Prof. Sandro Rigo (IC-Unicamp)