Análise Sintática

Guido Araújo
guido@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

Analisador Sintático (Parser)

- Receba uma seqüência de tokens do analisador léxico e determine se a string pode ser gerada através da gramática da linguagem fonte.
- É esperado ainda que ele reporte os erros de uma maneira inteligível
- Seja capaz de se recuperar de erros comuns, continuando a processar a entrada

Analisador Sintático (Parser)

- Seja capaz para expressar a sintaxe de linguagens de programação
  - Que tal dar nomes à ERs?

expr = ab(c|d)e

aux = c | d
expr = a b aux e

Context-free Grammars

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

- Como isso é implementado?
  - O analisador léxico substitui os nomes das ERs antes de traduzir para um autômato finito
  - sum = ([0-9]+ "+") * [0-9]+

Context-free Grammars

- 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

Context-free Grammars

- 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

O que está ocorrendo aqui?

Context-free Grammars

- 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
- Que notação usar para representar "palavras"(cadeias) dentro de uma frase?

Gramáticas Livre de Contexto

- Nomes não acrescentam a ERs o poder de expressar recursão.
- É isso que precisamos para expressar a recursão mútua entre sum e expr
- O que está faltando então?

Gramáticas Livre de Contexto

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

  symbol -> symbol symbol symbol … symbol

  Símbolos:

  - terminais: uma string do alfabeto da linguagem
  - não-terminais: aparecem do lado esquerdo
  - nenhum token aparece do lado esquerdo
  - existe um não-terminal definido como *start symbol*

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

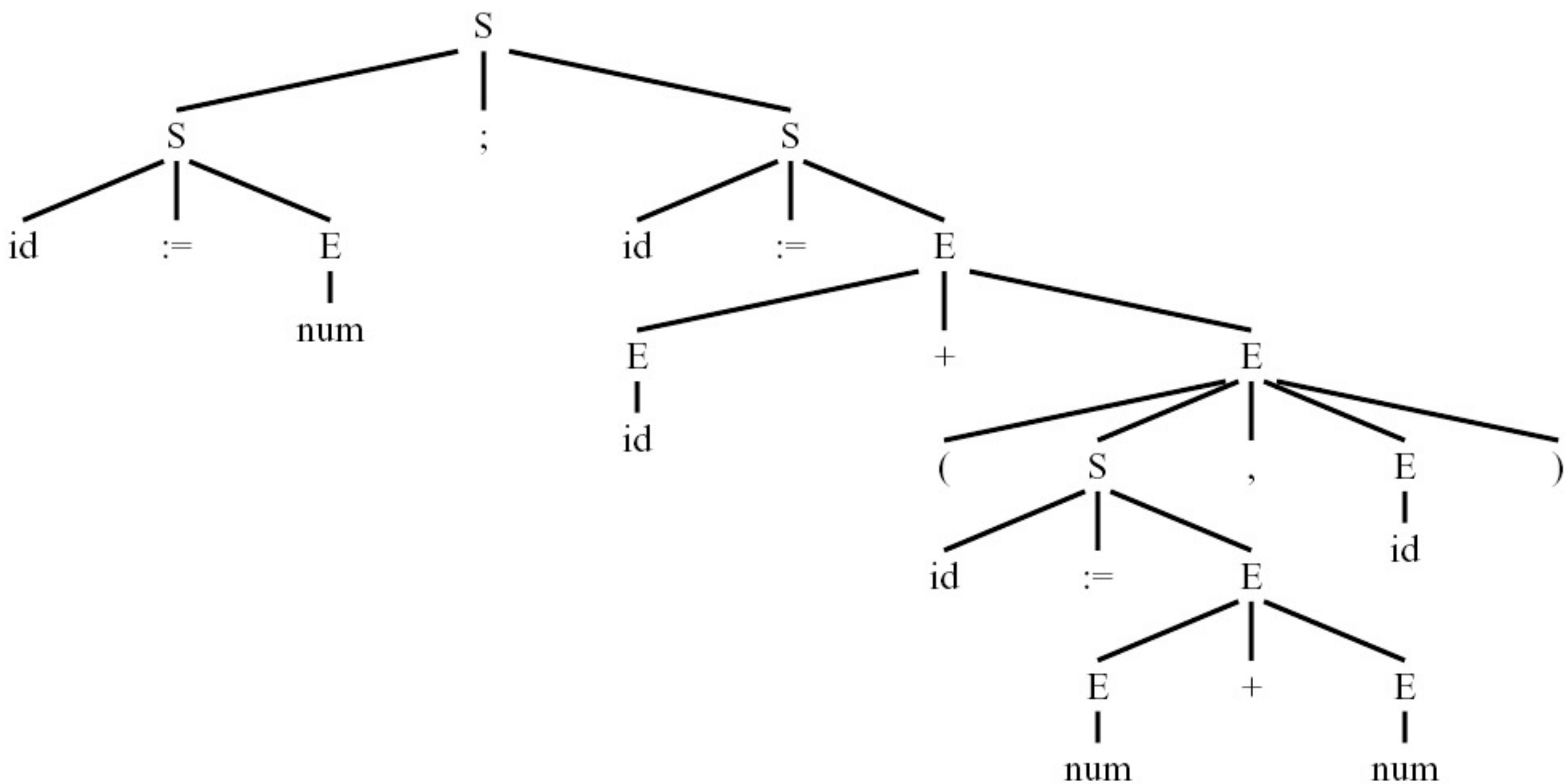
- *left-most:* o não terminal mais a esquerda é sempre o expandido;
- *right-most:* idem para o mais a direita.
- Qual é o caso do exemplo anterior?

Parse Trees

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

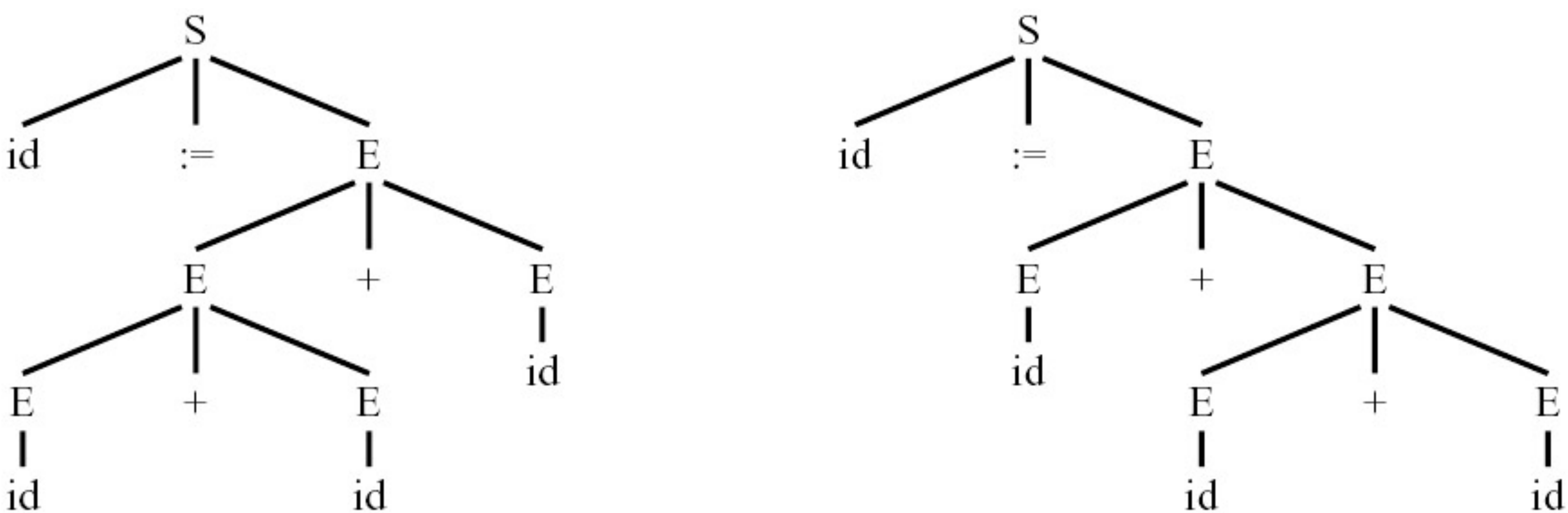
Parse Trees

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



Gramáticas Ambíguas

- Podemos derivar uma sentença com duas *parse trees* diferentes?
  - id := id+id+id



É Ambígua?

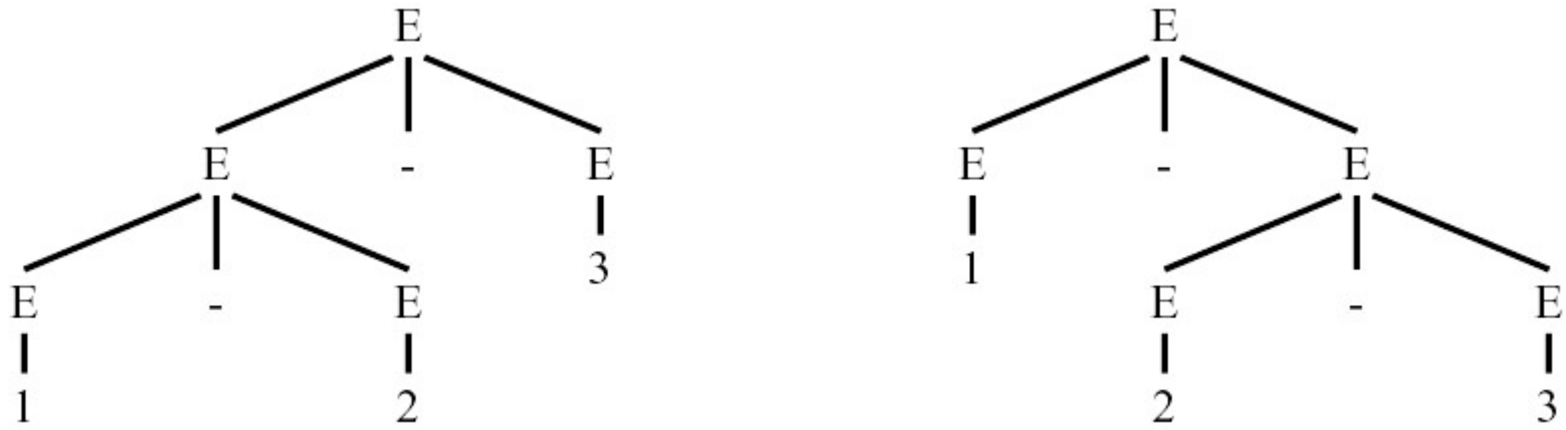
$E \rightarrow \text{id}$
$E \rightarrow \text{num}$
$E \rightarrow E * E$
$E \rightarrow E \,/\, E$
$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow (E)$

Construa Parse Trees para as seguintes expressões:

1-2-3

1+2*3

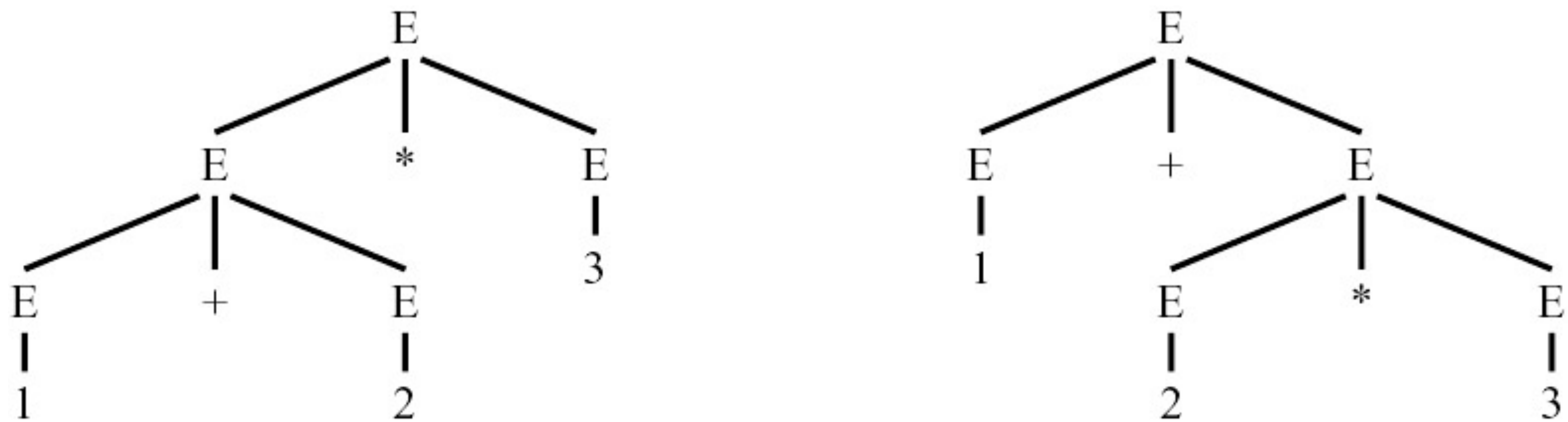
Exemplo: 1-2-3



Ambígua!

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

Exemplo: 1+2*3



Ambígua!

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

Mas qual o problema com isto?

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 a + 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

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \text{id}$$

$$E \rightarrow E - T \qquad T \rightarrow T \, / \, F \qquad F \rightarrow \text{num}$$

$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (E)$$

Construa as derivações e Parse Trees para as seguintes expressões:

1-2-3

1+2*3

Gramática para Expressões

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

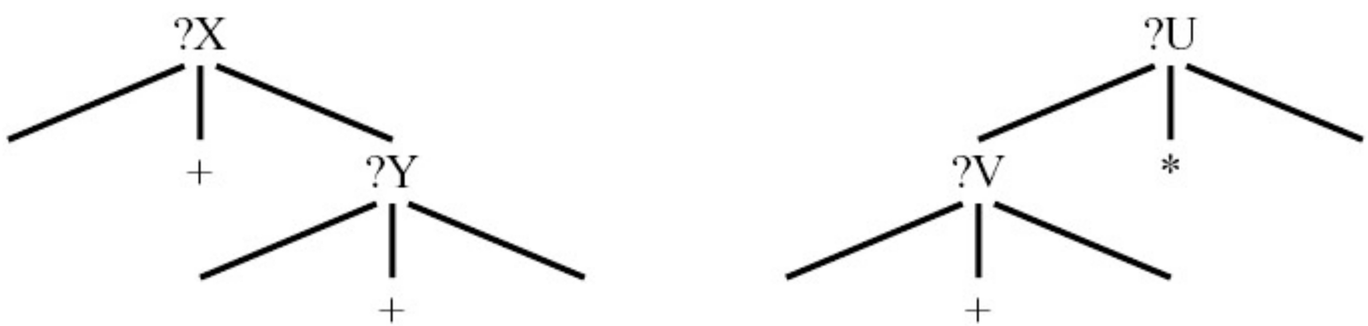
$T \rightarrow F$

$F \rightarrow \mathrm{id}$

$F \rightarrow \mathrm{num}$

$F \rightarrow (E)$

Essa gramática pode gerar as árvores abaixo?



Gramáticas Ambíguas

- Geralmente podemos trasformar 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

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

Criar um novo não terminal como símbolo inicial

Predictive Parsing

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

Predictive Parsing

$S \rightarrow if\ E\ then\ S\ else\ S$

$S \rightarrow begin\ S\ L$

$S \rightarrow print\ E$

$L \rightarrow end$

$L \rightarrow ;\ S\ L$

$E \rightarrow num = num$

Como seria um parser para essa gramática?

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); }
```

Predictive Parsing

$S \rightarrow E\ \$$

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \mathrm{id}$

$F \rightarrow \mathrm{num}$

$F \rightarrow (E)$

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

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 ?

E para 1*2-3 ?

FIRST and FOLLOW sets

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

Predictive Parsing

- Se uma gramática tem produções da forma:
  - X -> γ1
  - X -> γ2
    - Caso os conjuntos FIRST(γ1) e FIRST(γ2) 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 FIRST(XYZ) = FIRST (X) ?

Nullable

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

$Z \rightarrow d$

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

Nullable(Y)

Nullable(X)

Nullable(Z)

Nullable

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

FIRST

- FIRST(X) é o conjunto de terminais que podem iniciar strings derivadas de X

$Z \rightarrow d$

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

FIRST(Y)

FIRST(Z)

FIRST

- FIRST(*X*) é o conjunto de terminais que podem iniciar strings derivadas de *X*

$Z \rightarrow d$

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

FIRST(Y) = {c}

FIRST(Z) = {a,c,d}

FOLLOW

- FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X
  - *t* Є FOLLOW(*X*) se existe alguma derivação contendo *Xt*
  - Cuidado com derivações da forma X Y Z *t, onde Y e Z* podem ser vazios

$Z \rightarrow d$

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

FOLLOW(Y)

FOLLOW(Z)

FOLLOW

- FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X
  - t Є FOLLOW(X) se existe alguma derivação contendo Xt
  - Cuidado com derivações da forma X Y Z *t, onde Y e Z* podem ser vazios

$Z \rightarrow d$

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow$

$Y \rightarrow c$

$X \rightarrow Y$

$X \rightarrow a$

FOLLOW(Y) = {d,a,c}

FOLLOW(Z) = { }

Resumindo

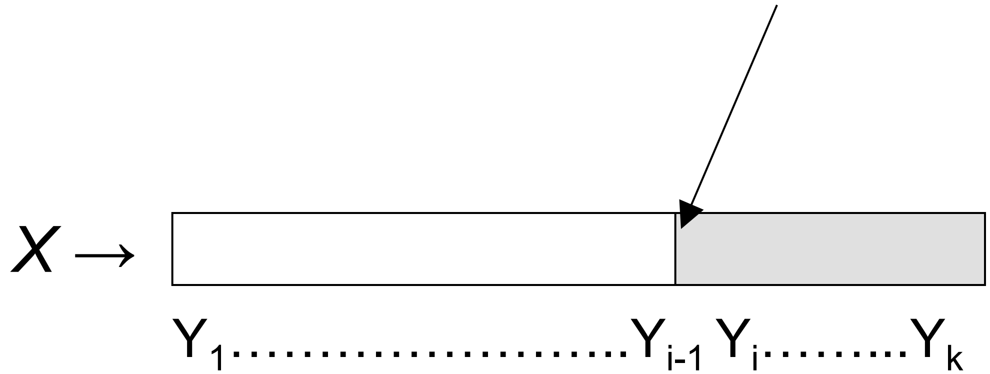
- Nullable(X) é verdadeiro se X pode derivar a string vazia
- FIRST(*γ*) é o conjunto de terminais que podem iniciar strings derivadas de *γ*
- FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X
  - *t* Є FOLLOW(*X*) se existe alguma derivação contendo *Xt*
  - Cuidado com derivações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

Definição FIRST, FOLLOW e nullable

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

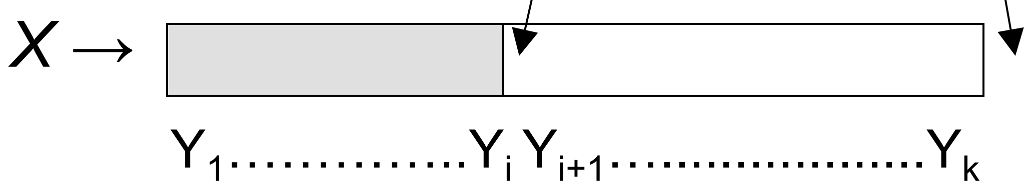
FIRST, FOLLOW, NULLABLE

$$\mathrm{FIRST}[X] \leftarrow \mathrm{FIRST}[X] \cup \mathrm{FIRST}[Y_i]$$



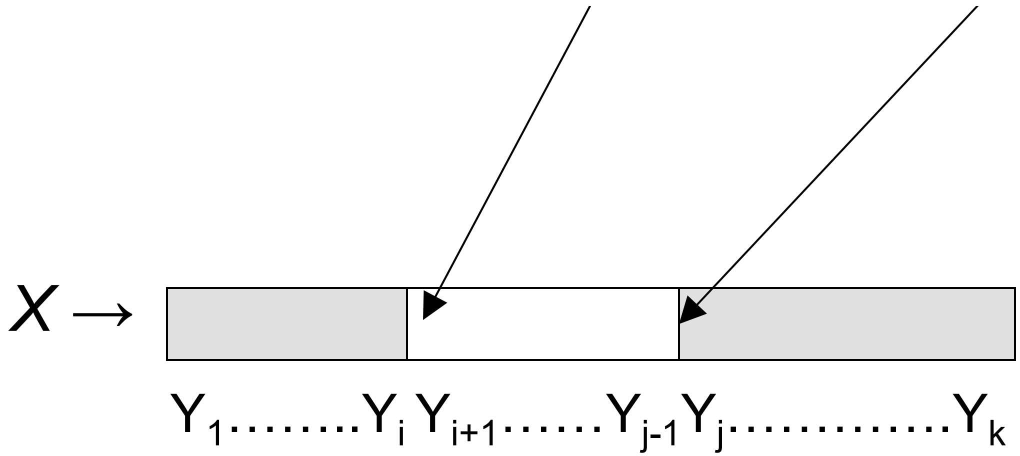
FIRST, FOLLOW, NULLABLE

$$\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FOLLOW}[X]$$



FIRST, FOLLOW, NULLABLE

$$\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FIRST}[Y_j]$$



Algoritmo FIRST, FOLLOW e nullable

- Algoritmo de iteração até um ponto fixo
- Os conjuntos poderiam ser computados de maneira separada
- Mesmo método usado para *Є-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			
Y			
Z			

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	a	c d
Y	yes	c	d
Z	no	d	

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

- 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*

Exemplo

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

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

$FIRST\ (\overset{\text{Y}}{XYZ})$

$FIRST\ (Y)$

Generalizando para strings

- FIRST($X\gamma$) = FIRST[X], if not nullable[X]
- FIRST($X\gamma$) = FIRST[X] U FIRST(γ), if nullable[X]
- string γ é *nullable* se cada símbolo em γ é *nullable*

Construindo um Predictive Parser

- Dada uma produção $X \rightarrow \gamma$
- Para cada t Є FIRST(γ)
  - 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 Є FOLLOW[X].

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	

Exemplo

- $Z \rightarrow d$ $\{d\}$
- $Z \rightarrow X\ Y\ Z$ $\{a,c,d\}$
- $Y \rightarrow$ $\{a,c,d\}$
- $Y \rightarrow c$ $\{c\}$
- $X \rightarrow Y$ $\{a,c,d\}$
- $X \rightarrow a$ $\{a\}$

	a	c	d
X			
Y			
Z			

Exemplo

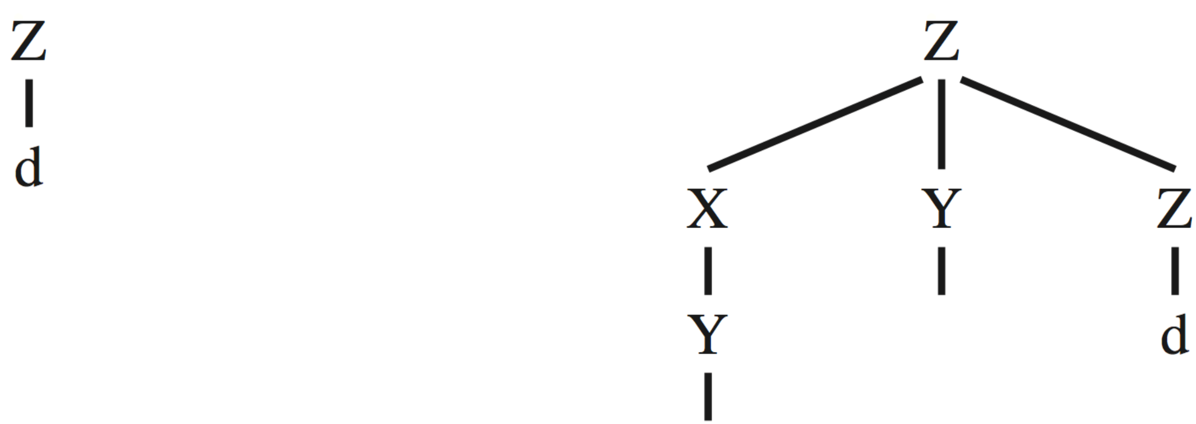
• $Z \rightarrow d$	{a,c,d}
• $Z \rightarrow X\ Y\ Z$	{a,c,d}
• $Y \rightarrow$	{a,c,d}
• $Y \rightarrow c$	{c}
• $X \rightarrow Y$	{a,c,d}
• $X \rightarrow a$	{a}

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

Recursão à Esquerda

$S \rightarrow E\ \$$

$E \rightarrow E - T$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

O que fazer para resolver?

Recursão à Esquerda

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

O que fazer para resolver?

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow$$

Recursão à direita!

Recursão à Esquerda

Generalizando:

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

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

Eliminando Recursão à Esquerda

- $S \rightarrow E\,\$$
- $E \rightarrow T\,E'$
- $E' \rightarrow +\,T\,E'$
- $E' \rightarrow -\,T\,E'$
- $E' \rightarrow$
- $T \rightarrow F\,T'$
- $T' \rightarrow *\,F\,T'$
- $T' \rightarrow /\,F\,T'$
- $T' \rightarrow$
- $F \rightarrow \text{id}$
- $F \rightarrow \text{num}$
- $F \rightarrow (E)$

	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

- $S \rightarrow E\,\$$
- $E \rightarrow T\,E'$
- $E' \rightarrow +\,T\,E'$
- $E' \rightarrow -\,T\,E'$
- $E' \rightarrow$
- $T \rightarrow F\,T'$
- $T' \rightarrow *\,F\,T'$
- $T' \rightarrow /\,F\,T'$
- $T' \rightarrow$
- $F \rightarrow \text{id}$
- $F \rightarrow \text{num}$
- $F \rightarrow (E)$

	+	*	id	(	)	\$
S			$S \rightarrow E\$$	$S \rightarrow E\$$		
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 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);
 }}
```