Geração de Parser

Predictive Parsing

Hervé Yviquel
herve@ic.unicamp.br

Universidade Estadual de Campinas (Unicamp)
Instituto de Computação (IC)
Laboratório de Sistemas de Computação (LSC)

MC921 • Projeto e Construção de Compiladores • 2023 S2

Aula Anterior

Resumo

- Geração do Lexer
  - a partir da especificação dos tokens com expressões regulares
- Autômato Finito Não-determinístico
- Autômato Finito Determinístico

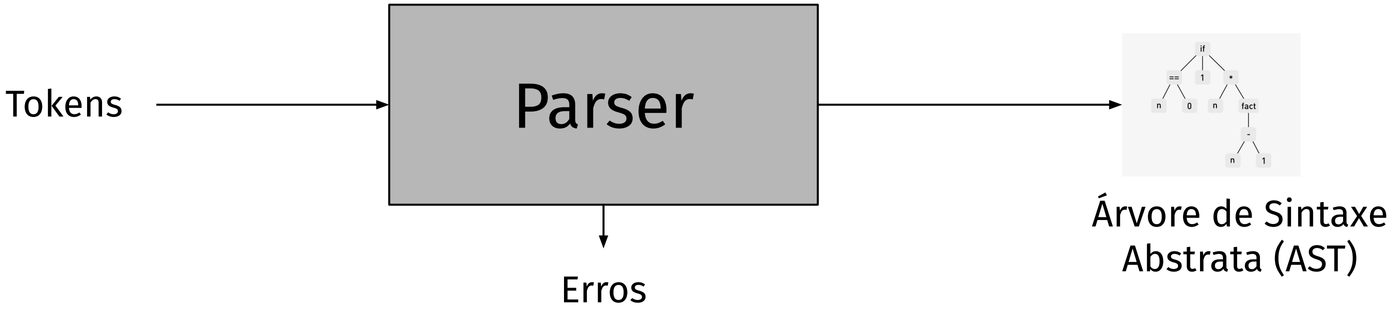
Aula de Hoje

Plano

- Gerar um Parser
- Parser Preditivo
- Conjuntos FIRST e FOLLOW
- Construção do Parser Preditivo
- Ambiguidade
- Recuperação de Erros

Gerar um Parser

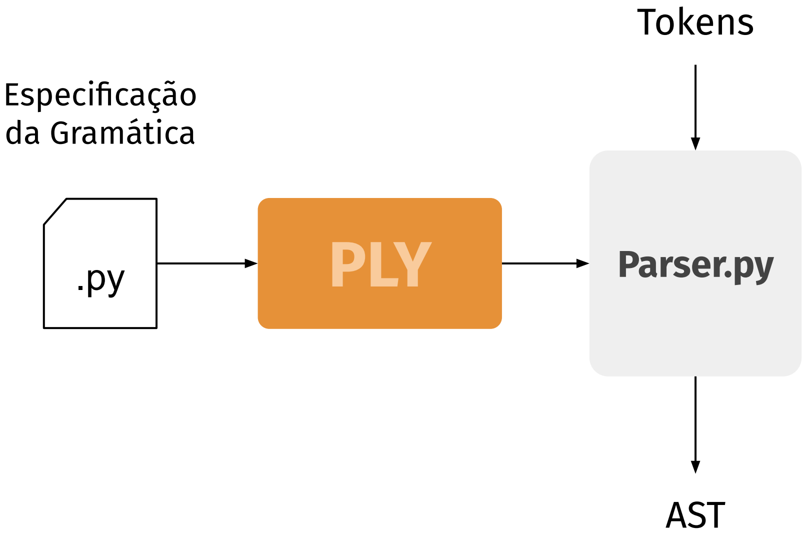
Analisador Sintático



- Recebe uma sequência de tokens e determina se 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

Fluxo do gerador de parser

- O gerador de parser gera o parser automaticamente
  - a partir da especificação
- Tem várias geradores de parser disponíveis
  - Yacc, Bison, ANTLR, PLY, ...



Parser Preditivo

Predictive Parsing

- Também chamados de recursive-descent
  - ou ainda parser LL
- 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

Vamos olhar um exemplo...

S → if E then S else S

S → begin S L

S → print E

L → end

L → ; S L

E → num = num

Como seria um parser para essa gramática?

Implementação do Parser


```
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); }
```

Outro exemplo

S → E $

E → E + T

E → E - T

E → T

T → T * F

T → T / F

T → F

F → id

F → num

F → (E)

- Vamos aplicar a mesma técnica para essa outra gramática...
- Problema: Quando escolher regras para E e para T?
  - 1*2+3 ?
  - 1*2-3 ?

Implementação do Parser


```
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(); }}
```

S → E $
E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F
F → id
F → num
F → (E)

Gramática suportada

- Parsers preditivos apenas funcionam quando o primeiro símbolo terminal permite escolher a regra da gramática
    - senão a gramática não pode ser analisada com parser preditivo
- Vamos procurar os primeiros símbolos para cada regra
    - de forma automática usando um algoritmo

Conjuntos FIRST e FOLLOW

Conjunto FIRST

- 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 → T * F

γ = T * F

FIRST(γ) = ?

FIRST(γ) = {id ,num, (}

```
S → E $
E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F
F → id
F → num
F → (E)
```

Gramática suportada

- Se uma gramática tem produções da forma:

  X -> γ1
  X -> γ2

- Caso FIRST(γ1) e FIRST(γ2) tenham intersecção
  - a gramática não pode ser analisada com um predictive parser
- Por que?
  - A função recursiva não vai saber que caso executar

Calculando FIRST

- Z → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → a

- Como seria para Z → X Y Z ?
- Podemos simplesmente fazer FIRST(XYZ) = FIRST(X) ?

Nullable

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

- Z → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → 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 → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → a

FIRST(Y) = {c}

FIRST(X) = {a,c}

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

FOLLOW

FOLLOW(X) é o conjunto de terminais que podem imediatamente seguir X

- Z → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → a

t Є FOLLOW(A) se existe alguma derivação contendo A t

Cuidado com derivações na forma B A t, onde A pode ser vazio. Neste caso FOLLOW(B) também contém t.

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

FOLLOW(Z) = { }

Resumindo

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

Construção do Parser Preditivo

Definição FIRST, FOLLOW e nullable


```
for each terminal symbol Z
    FIRST[Z] ← {Z}
repeat
    for each production X → Y1 Y2 ··· Yk
        if Y1 ... Yk are all nullable (or if k = 0)
          then nullable[X] ← true
        for each i from 1 to k, each j from i + 1 to k
            if Y1 ··· Y(i−1) are all nullable (or if i = 1)
              then FIRST[X] ← FIRST[X] ∪ FIRST[Yi]
            if Y(i+1) ··· Yk are all nullable (or if i = k)
              then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FOLLOW[X]
            if Y(i+1) ··· Y(j−1) are all nullable (or if i + 1 = j)
              then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FIRST[Yj]
until FIRST, FOLLOW, and nullable did not change in this iteration.
```

- Algoritmo de iteração até um ponto fixo
- Os conjuntos poderiam ser computados de maneira separada

Definição FIRST, FOLLOW e nullable

for each terminal symbol Z
  FIRST[Z] $\leftarrow \{Z\}$
repeat
  for each production $X \to Y_1 Y_2 \cdots Y_k$
    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.

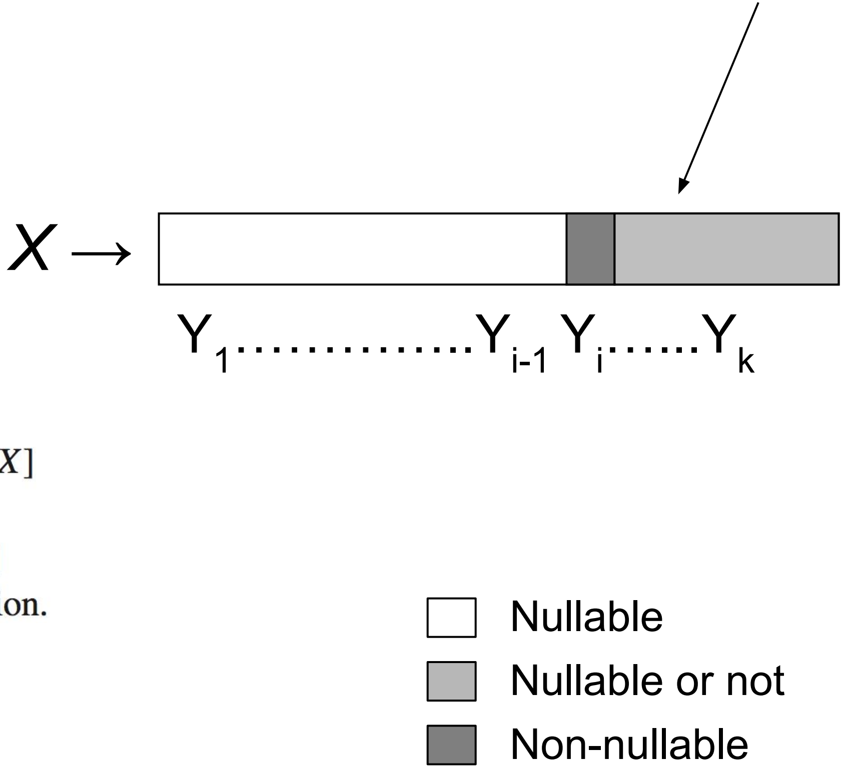
Definição FIRST, FOLLOW e nullable

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.

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.

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



Definição FIRST, FOLLOW e nullable

for each terminal symbol Z

  $\mathrm{FIRST}[Z] \leftarrow \{Z\}$

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 $\mathrm{nullable}[X] \leftarrow \mathrm{true}$

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

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

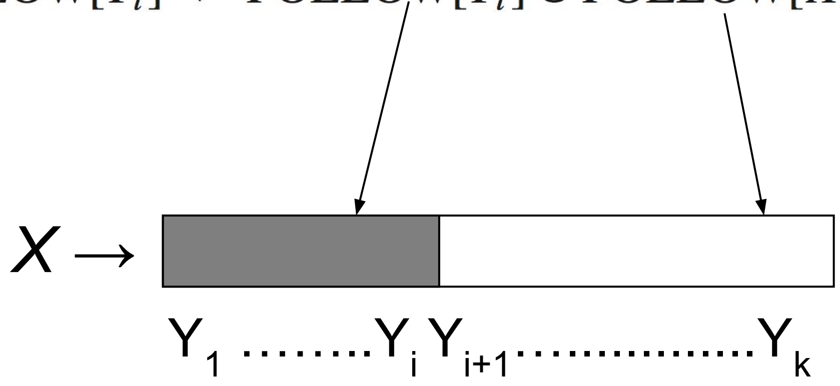
        then $\mathrm{FOLLOW}[Y_i] \leftarrow \mathrm{FOLLOW}[Y_i] \cup \mathrm{FOLLOW}[X]$

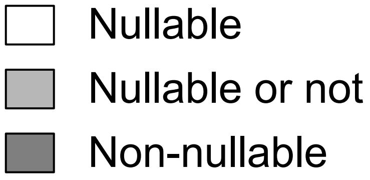
      if $Y_{i+1} \cdots Y_{j-1}$ are all nullable (or if $i + 1 = j$)

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

until FIRST, FOLLOW, and nullable did not change in this iteration.

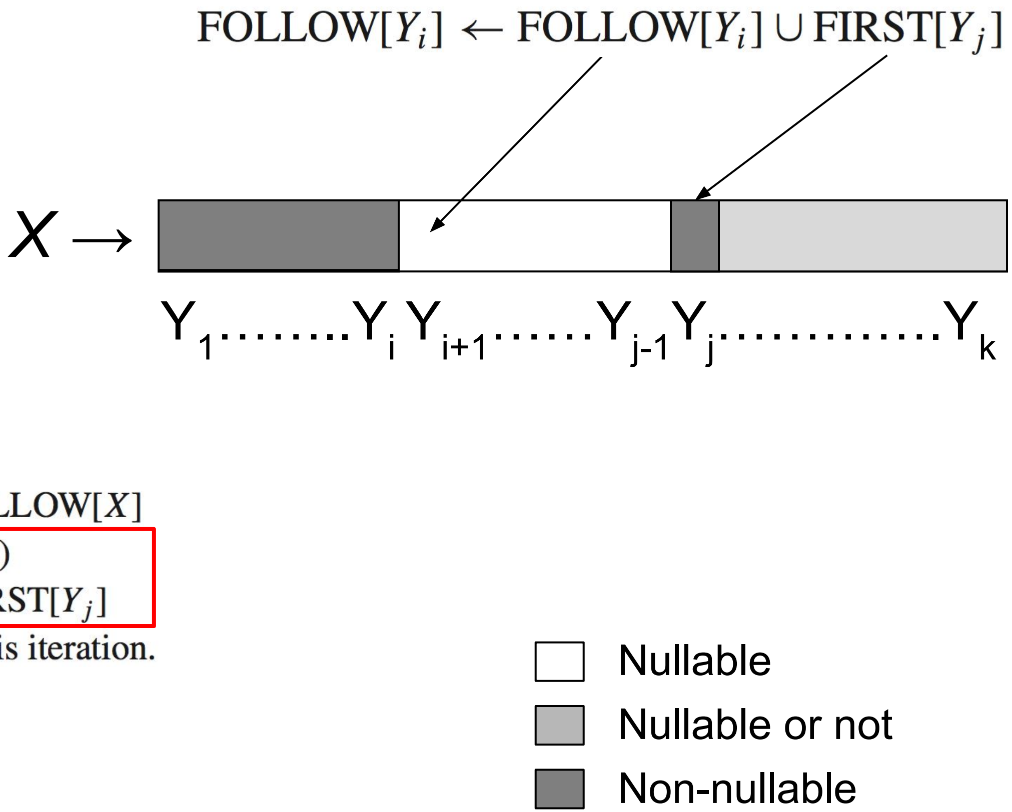
$$\mathrm{FOLLOW}[Y_i] \leftarrow \mathrm{FOLLOW}[Y_i] \cup \mathrm{FOLLOW}[X]$$





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.



Exemplo

- Z → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → a

	nullable	FIRST	FOLLOW
X			
Y			
Z			

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.

Exemplo

- Z → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → a

	nullable	FIRST	FOLLOW
X	no	{ }	{ }
Y	no	{ }	{ }
Z	no	{ }	{ }

for each terminal symbol Z
    $\text{FIRST}[Z] \leftarrow \{Z\}$
repeat
    for each production $X \to Y_1 Y_2 \cdots Y_k$
        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.

Exemplo

- Z → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → a

	nullable	FIRST	FOLLOW
X	no	{ a }	{ d, c }
Y	yes	{ c }	{ d }
Z	no	{ d }	{ }

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

Exemplo

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

	nullable	FIRST	FOLLOW
X	yes	{ a, c }	{ a, d, c }
Y	yes	{ c }	{ a, c, d }
Z	no	{ a, c, d }	{ }

for each terminal symbol Z
  FIRST[Z] $\leftarrow \{Z\}$
repeat
  for each production $X \to Y_1 Y_2 \cdots Y_k$
    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.

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

- Z → d
- Z → X Y Z
- Y →
- Y → c
- X → Y
- X → a

γ

FIRST(XYZ)

γ

FIRST(Y)

	nullable	FIRST	FOLLOW
X	yes	{ a, c }	{ a, d, c }
Y	yes	{ c }	{ a, c, d }
Z	no	{ a, c, d }	{ }

Generalizando cálculo de γ

- Assuma que γ = Aβ
- FIRST(Aβ) = FIRST[A], if not nullable[A]
- FIRST(Aβ) = FIRST[A] U FIRST(β), if nullable[A]

Calculando FIRST(γ) onde A -> γ

	nullable	FIRST	FOLLOW
X	yes	{ a, c }	{ a, d, c }
Y	yes	{ c }	{ a, c, d }
Z	no	{ a, c, d }	{ }

	FIRST(γ)		FIRST(γ)
• Z → d	{ }	• Y → c	{ }
• Z → X Y Z	{ }	• X → Y	{ }
• Y →	{ }	• X → a	{ }

Calculando FIRST(γ) onde A $\longrightarrow \gamma$

	nullable	FIRST	FOLLOW
X	yes	{ a, c }	{ a, d, c }
Y	yes	{ c }	{ a, c, d }
Z	no	{ a, c, d }	{ }

	FIRST(γ)		FIRST(γ)
• Z → d	{ d }	• Y → c	{ c }
• Z → X Y Z	{ a, c, d }	• X → Y	{ c }
• Y →	{ }	• X → a	{ a }

Construindo um Predictive Parser Table

Dada uma produção X →γ

1. Para cada t ∈ FIRST(γ)
   - Coloque a produção X → γ na linha X, coluna t.
2. Se γ é nullable
   - Coloque a produção na linha X, coluna t para cada t ∈ FOLLOW[X].

Exemplo

	a	c	d
X			
Y			
Z			

Exemplo

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

Inserindo $X \rightarrow \gamma$ para todo $t \in$ FIRST(γ)

	a	c	d
X			
Y			
Z			

Exemplo

• Z → d	{ d }	• Y → c	{ c }
• Z → X Y Z	{ a, c, d }	• X → Y	{ c }
• Y →	{ }	• X → a	{ a }

Inserindo $X \rightarrow \gamma$ para todo t Є FIRST(γ)

	a	c	d
X	X → a	X → Y	
Y		Y → c	
Z	Z → X Y Z	Z → X Y Z	Z → d Z → X Y Z

Exemplo

• Z → d	{ d }	• Y → c	{ c }
• Z → X Y Z	{ a, c, d }	• X → Y	{ c }
• Y →	{ }	• X → a	{ a }

Inserindo $X \rightarrow \gamma$ para todo *t* Є FIRST(γ)

Inserindo Y→ para todo *t* Є FOLLOW[Y] = {a,c,d}

	a	c	d
X	X → a	X → Y	
Y		Y → c	
Z	Z → X Y Z	Z → X Y Z	Z → d Z → X Y Z

Exemplo

• Z → d	{ d }	• Y → c	{ c }
• Z → X Y Z	{ a, c, d }	• X → Y	{ c }
• Y →	{ }	• X → a	{ a }

Inserindo *X* →γ para todo *t* Є FIRST(γ)

Inserindo Y→ para todo *t* Є FOLLOW[Y] = {a,c,d}

	a	**c**	**d**
X	X → a	X → Y	
Y	Y →	Y → c Y →	Y →
Z	Z → X Y Z	Z → X Y Z	Z → d Z → X Y Z

Exemplo

• Z → d	{ d }	• Y → c	{ c }
• Z → X Y Z	{ a, c, d }	• X → Y	{ c }
• Y →	{ }	• X → a	{ a }

Inserindo $X \rightarrow \gamma$ para todo *t* Є FIRST(γ)

Inserindo Y→ para todo *t* Є FOLLOW[Y] = {a,c,d}

Inserindo X→Y para todo *t* Є FOLLOW[X] = {a,c,d}

	a	c	d
X	X → a X → Y	X → Y	X → Y
Y	Y →	Y → c Y →	Y →
Z	Z → X Y Z	Z → X Y Z	Z → d Z → X Y Z

Exemplo

	a	c	d
X	X → a X → Y	X → Y	X → Y
Y	Y →	Y → c Y →	Y →
Z	Z → X Y Z	Z → X Y Z	Z → d Z → X Y Z

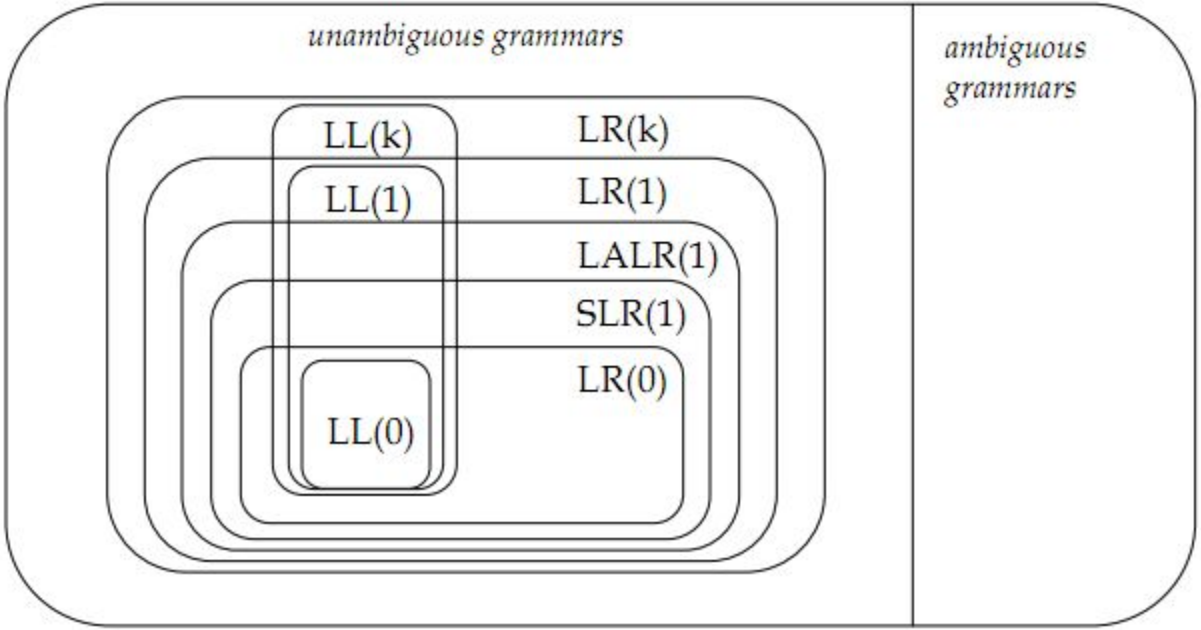
Não funciona!

Parsers Preditivos

- 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
    - Por exemplo, LL(2)
    - Gera uma tabela onde as linhas são os não-terminais e as colunas são todas as sequências possíveis de 2 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!

LL(1) versus LR(k)

A picture is worth a thousand words:



Ambiguidade

Recursão à Esquerda


```
S → E $
E → E - T
E → E + T
E → T
T → T * F
T → T / F
T → F
F → id
F → num
F → (E)
```

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

Recursão à Esquerda


```
S → E $
E → E - T
E → E + T
E → T
T → T * F
T → T / F
T → F
F → id
F → num
F → (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 → Xγ e X → α, onde α não começa com X
- Derivamos strings da forma αγ*
  - α seguido de zero ou mais γ
- Podemos reescrever:

$$\left(\begin{array}{lll} X & \rightarrow & X\ \gamma_1 \\ X & \rightarrow & X\ \gamma_2 \\ X & \rightarrow & \alpha_1 \\ X & \rightarrow & \alpha_2 \end{array}\right) \Longrightarrow \left(\begin{array}{lll} X & \rightarrow & \alpha_1\ X' \\ X & \rightarrow & \alpha_2\ X' \\ X' & \rightarrow & \gamma_1\ X' \\ X' & \rightarrow & \gamma_2\ X' \\ X' & \rightarrow & \end{array}\right)$$

Eliminando Recursão à Esquerda

- S → E $
- E → T E ′
- E ′ → + T E ′
- E ′ →- T E ′
- E ′ →
- T → F T ′
- T ′ →* F T ′
- T ′ → / F T ′
- T ′ →
- F → id
- F → num
- F → (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 → E $
- E → T E ′
- E ′ → + T E ′
- E ′ →- T E ′
- E ′ →
- T → F T ′
- T ′ →* F T ′
- T ′ → / F T ′
- T ′ →
- F → id
- F → num
- F → (E)

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

Recuperação de Erros

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 (Assume)


```
void T() {

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

}
```

Recuperação de Erros (Recover)


```
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);
    }
}
```

Exercícios

Exercícios

a. Calculate nullable, FIRST, and FOLLOW for this grammar:

$$
\begin{aligned}
S &\to u\ B\ D\ z \\
B &\to B\ v \\
B &\to w \\
D &\to E\ F \\
E &\to y \\
E &\to \\
F &\to x \\
F &\to
\end{aligned}
$$

b. Construct the LL(1) parsing table.

3.6

u $S \to uB \mid Dz$

w $B \to Bv$

w $B \to w$

x, y, z $D \to EF$

y $E \to y$

x, z $E \to$

x $F \to x$

z $F \to$

LEFT-RECURSION

w $B \to wB'$

v $B' \to vB'$

x, y, z $B' \to$

(b) BUILD LL(1)

	FIRST	FOLLOW	NULL.
S	u	\$	
B	w	~~v,~~ x, y, z	
D	y, x	z	✓
E	y	x, z	✓
F	x	z	✓

Resumo

- Gerar um Parser
- Parser Preditivo
- Conjuntos FIRST e FOLLOW
- Construção do Parser Preditivo
- Ambiguidade
- Recuperação de Erros

Leitura Recomendada

- Capítulo 3.1-3.2 do livro do Cooper.
- Capítulo 3-3.1 do livro do Appel.

Próxima Aula

- Geração do Parser
  - Parser LR

Obrigado!
Merci!

UNICAMP

Pallete

Bubbles

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

DRACULA

Tables

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Emojis

👍 👎 ⚠️ ✅ ❌

🔁 🆗 🆕 ⬆️ ⬇️