

Análise sintática descendente

1 de junho de 2024

18:05

https://files.sofiars.xyz/slides/parsing_ll/

Ferramentas como o Ply usam parsing ascendente (ou *bottom-up*).

Gramática exemplo:

r1 : $S \rightarrow 'a' X 'c'$

r2 : $X \rightarrow 'b' X$

r3 : $X \rightarrow \epsilon$

ou

r1 : $S \rightarrow 'a' X 'c'$

r2 : $X \rightarrow 'b' X \mid \epsilon$

Stack	Input	Ação
	a b b c \$	Shift a
a	b b c \$	Shift b
a b	b c \$	Shift b
a b b	c \$	Reduce r3
a b b X	c \$	Reduce r2
a b X	c \$	Reduce r2
a X	c \$	Shift c
a X c	\$	Reduce r1
S	\$	

O parsing descendente (ou *top-down*) funciona de forma oposta.

Stack	Input	Ação
S	a b b c \$	Expandir r1
a X c	a b b c \$	Prox. token
X c	b b c \$	Expandir r2
b X c	b b c \$	Prox. token
X c	b c \$	Expandir r2
b X c	b c \$	Prox. token
X c	c \$	Expandir r3
c	c \$	Prox. token

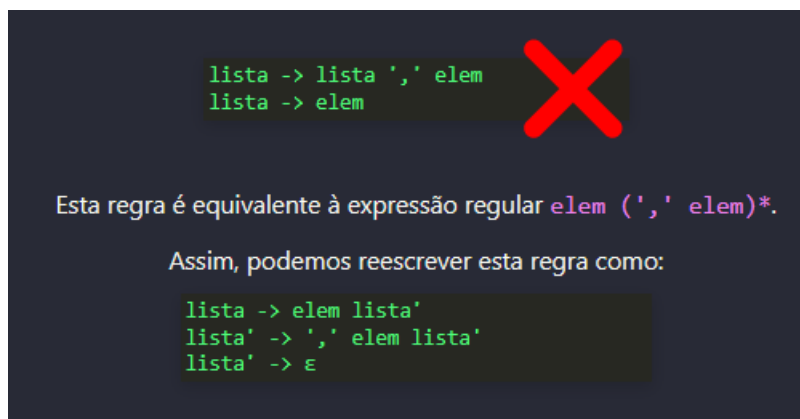
	\$	
--	----	--

- No parsing ascendente, reduzimos os símbolos de entrada, usando as regras da gramática, até chegar ao símbolo inicial.
- No parsing descendente, aplicamos as regras da gramática, começando pelo símbolo inicial, até chegar aos símbolos de entrada.

Os parsers descendentes apenas permitem analisar gramáticas LL, isto é, gramáticas nas quais apenas usamos 1 símbolo para lookahead em cada passo.

O que é que distingue uma gramática LL(1)?

1. Não pode apresentar recursividade à esquerda (caso contrário, o parser entraria num ciclo infinito).



```

lista -> lista ',' elem
lista -> elem

```

Esta regra é equivalente à expressão regular `elem (',' elem)*`.

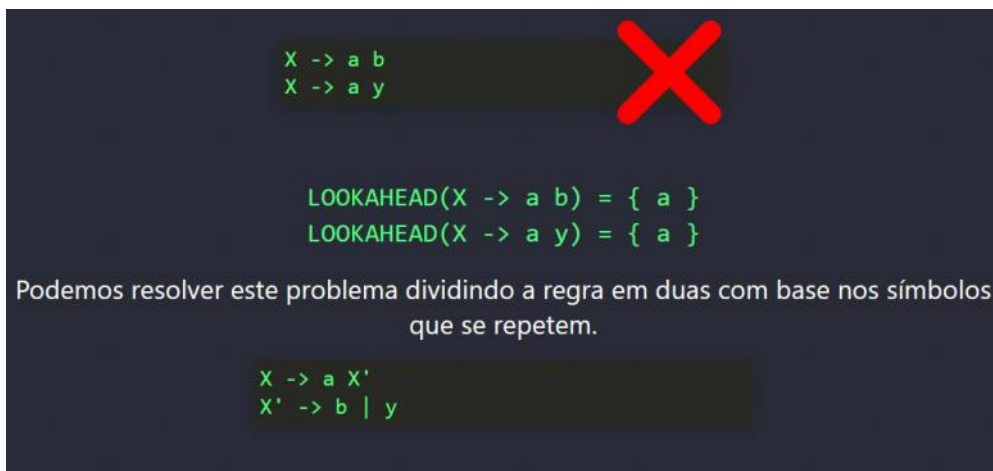
Assim, podemos reescrever esta regra como:

```

lista -> elem lista'
lista' -> ',' elem lista'
lista' -> ε

```

2. Regras com o mesmo símbolo à esquerda não podem ter lookaheads que se intersejam



```

X -> a b
X -> a y

```

`LOOKAHEAD(X -> a b) = { a }`
`LOOKAHEAD(X -> a y) = { a }`

Podemos resolver este problema dividindo a regra em duas com base nos símbolos que se repetem.

```

X -> a X'
X' -> b | y

```

Como calcular o lookahead?

Primeiro, é necessário entender dois conceitos, FIRST e FOLLOW.

1. FIRST - Para um dado símbolo A, $\text{FIRST}(A)$ é o conjunto de símbolos terminais que iniciam as sequências derivadas a partir de A.

- Se A for um símbolo terminal, $\text{FIRST}(A) = A$.
- No caso de uma produção do tipo $A \rightarrow b C$, se b derivar ϵ , incluímos também $\text{FIRST}(C)$ em $\text{FIRST}(A)$. (b é ignorado).

Por exemplo:

$E \rightarrow F + E \mid F$
 $F \rightarrow \epsilon$
 $F \rightarrow (E)$

c.

$\text{FIRST}(F) = \text{FIRST}(\epsilon) + \text{FIRST}((E)) = \{ \epsilon, (\}$
 $\text{FIRST}(E) = \text{FIRST}(F + E) + \text{FIRST}(F)$
 $\text{FIRST}(E) = \text{FIRST}(F) + \text{FIRST}(+) = \{ \epsilon, (, + \}$

2. FOLLOW - Sendo A um símbolo não terminal, $\text{FOLLOW}(A)$ é o conjunto de símbolos terminais que podem aparecer imediatamente a seguir a A.

(Ignorando derivações de A)

Por exemplo:

$E \rightarrow F + E \mid F$
 $F \rightarrow \epsilon$
 $F \rightarrow (E)$

a.

$\text{FOLLOW}(E) = \{ \$,) \}$
 $\text{FOLLOW}(F) = \{ + \} \cup \text{FOLLOW}(E)$
 $\text{FOLLOW}(F) = \{ +, \$,) \}$

$E \mapsto (F) \rightarrow \text{deriva em } \{ \$,) \}$
 $F \mapsto (+ E) \rightarrow + \cup \uparrow$
 $= \{ +, \$,) \}$

Agora já se pode calcular o conjunto LOOKAHEAD das produções da

gramática.

LOOKAHEAD(A \rightarrow a) = FIRST(a) se a não deriva ϵ .
LOOKAHEAD(A \rightarrow a) = FIRST(a) U FOLLOW(A) caso contrário.

```
E  $\rightarrow$  F + E | F
F  $\rightarrow$   $\epsilon$ 
F  $\rightarrow$  ( E )
```

LOOKAHEAD(E \rightarrow F + E) = FIRST(F + E) = { ϵ , (, + }
LOOKAHEAD(E \rightarrow F) = FIRST(F) U FOLLOW(E) = { ϵ , (, \$,) }
LOOKAHEAD(F \rightarrow ϵ) = FIRST(ϵ) U FOLLOW(F) = { ϵ , +, \$,) }
LOOKAHEAD(F \rightarrow (E)) = FIRST((E)) = { (}

Existe uma interseção entre duas regras com o símbolo E à esquerda, logo a gramática não é LL(1).