Lecture 7

Top Down Parsing

Derive a string matching a source string through a sequence of **derivations starting with distinguished (starting) symbol**

Starts with the root of the parse tree

For valid source string α, a top down parse determines a derivation sequence

$$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \alpha$$

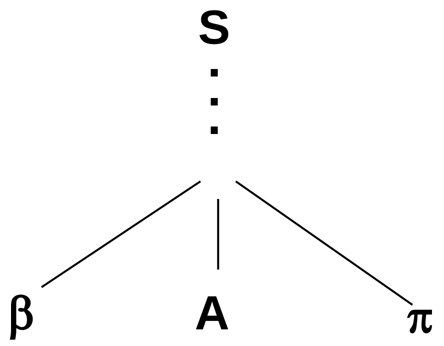
Steps in top down parsing

1. Current sentential form (CSF) := ‘S’

2. Let CSF be of the form **$\beta A\pi$**, s.t. β is string of Ts (may be null), A is leftmost NT in CSF. **Exit with success if CSF := α**

3. Make a derivation **$A \Rightarrow \beta_1 B\delta$** according to production A::= $\beta_1 B\delta$ of G s.t. β_1 is string of Ts (may be null). This makes **CSF=$\beta\beta_1 B\delta\pi$**

4. Goto step 2

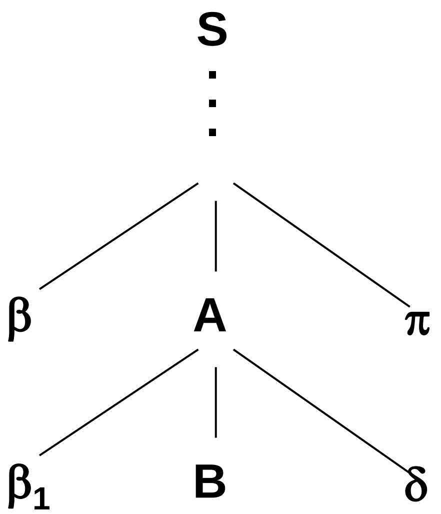


$\beta A \pi$

CSF

Parse tree

(a)



$\beta\beta_1 B\delta\pi$

CSF

Parse tree

(b)

Top down parsing is also known as **left-to-left parsing (LL parsing)** [since derivation for leftmost NT)

Let **CSF** $\equiv \gamma$**C**δ with C as leftmost NT in it and let grammar production for C be

$$\mathbf{C ::= \rho \mid \sigma}$$

where each of ρ, σ is a string of terminal and non terminal symbols

#	Production rule
1	expr → expr + term
2	| expr - term
3	| term
4	term → term * factor
5	| term / factor
6	| factor
7	factor → number
8	| identifier

Input String: x – 2 * y

Rule	Sentential form	Input string
-	expr	↑ x – 2 * y
2	expr – term	↑ x – 2 * y
3	term – term	↑ x – 2 * y
6	factor – term	↑ x – 2 * y
8	id - term	x ↑ – 2 * y
-	id, x - term	x - ↑ 2 * y
4	id, x - term * factor	x - ↑ 2 * y
6	id, x - factor * factor	x - 2 ↑ * y
7	id, x - num * factor	x - 2 ↑ * y
-	id, x - num, 2 * factor	x - 2 * ↑ y
8	id, x - num, 2 * id	x - 2 * y↑
8	id, x - num, 2 * id, y	x - 2 * y↑

Which RHS alternative should parser choose?

Alternative chosen may lead to a string of Ts which does not match with source string α

In such cases, other alternatives would have to be tried out until we **derive a sentence that matches source string**, or until we have **generated all possible sentences w/o obtaining a sentence** that matches source string

One approach would be generating complete sentences of source language and compare them with α to check if match exists. **<u>This is inefficient</u>**

Introduce a check called **continuation check,** to **determine whether current sequence of derivations may be able to find a successful parse of α**

Continuation Check

1. Let CSF be of the form $\beta A\pi$, β is string of n Ts
2. **For successful parse, β must match the first n symbols of α**
3. Apply this check at every step, and abandon the current sequence of derivations if violated

Incremental continuation check

Continuation check may be applied **incrementally** as follows

1. Let CSF $\equiv \beta A \pi$, then source string must be $\beta\ldots$
2. If prediction for A is $A \Rightarrow \beta_1 B \delta$, where β_1 is string of m terminal symbols, then string **β_1 must match m symbols following β in source string**
3. Compare β_1 with m symbols to the right of β in source string

Incremental check is more economical than continuation check which compares $\beta\beta_1$ with **(n+m)** symbols in source string

Predictions and Backtracking

$$\text{CSF} \equiv \beta A \pi$$

$$\begin{array}{ll} & \quad\;\; \text{SSM} \\ & \quad\;\;\; \downarrow \\ \text{Source string :} & \beta \;\; t \end{array}$$

Where CSF $\equiv \beta A\pi$ implies S $\Rightarrow \beta A\pi$ and **source string marker (SSM) points at first symbol following β in source string** i.e. terminal symbol t

Parsing proceed as follows

1. Identify leftmost NT in CSF, i.e. A

2. Select an alternative on RHS of production for A

3. **Since we do not know whether derived string will satisfy continuation check**, call this choice a **<u>prediction</u>**

4. This continuation check is applied incrementally to terminal symbol (s), if any, occurring in leftmost position of prediction

SSM is incremented if check succeeds and parsing continues

If check fails, one or more predictions are discarded and SSM is reset to its value before rejected prediction was made

This is called **backtracking**

Example :   id + id * id

Production :

E := T + E | T
T := V * T | V
V := id

1. SSM = 1, CSF = E
2. Make prediction E → T + E. CSF = T + E
3. Make prediction T → V * T. CSF = V * T + E
4. Make prediction V → id. CSF = id * T + E. id matches first symbol of string, hence SSM = SSM + 1
5. Match second symbol, *, this match fails hence reject
   T → V * T. Reset SSM =1, CSF = T + E
6. Make new prediction for T, T → V. CSF = V + E
7. Make prediction V → id. CSF = id + E.
   id matches first symbol of string,
   hence SSM = SSM + 1

8. Match second symbol of prediction + . Match succeeds, hence SSM = SSM + 1
9. Make prediction E → T + E. CSF = id + T + E
10. Make prediction T → V * T. CSF = id + V * T + E
11. ……. So on

Example

Prediction	Predicted sentential form
$E \Rightarrow T + E$	$T + E$
$T \Rightarrow V$	$V + E$
$V \Rightarrow id$	$id + E$
$E \Rightarrow T$	$id + T$
$T \Rightarrow V * T$	$id + V * T$
$V \Rightarrow id$	$id + id * T$
$T \Rightarrow V$	$id + id * V$
$V \Rightarrow id$	$id + id * id$

Top down parsing w/o backtracking

Achieved as follows

1. If leftmost NT is A and source symbol pointed to by SSM is ‘t’, parser selects that RHS alternative of A which can produce ‘t’ as its first terminal symbol

2. An error is signaled if no RHS alternative can produce ‘t’ as first terminal symbol

Consider parsing of string **id+id*id**
First prediction is to be made using prod

E ::= T + E | T

s.t. the first terminal symbol produced by it would be id, first symbol in source string

From grammar, we find that **T ⇒ V … and V ⇒ id**
Thus, **any RHS alternative can produce id in first position**

Both alternative of E start with T

Use **left factoring** to ensure that RHS alternatives will **produce unique terminal symbols in first position**

Production for E

E ::= T + E | T

rewritten as

E ::= TE''

E'' ::= + E | ε

First prediction is E $\Rightarrow$ TE'' since first source symbol is 'id'

When E’’ becomes leftmost NT in CSF, we would make prediction E’’ $\Rightarrow$ +E if next source symbol is ‘+’ and prediction E’’ $\Rightarrow$ E or E’’ $\Rightarrow \varepsilon$ in all other cases
Such parsers are called **predictive parsers**

Complete rewritten form is

E ::= TE’’
E’’ ::= + E | ε
T ::= VT’’
T’’ ::= *T | ε
V ::= id

Parsing of id+id*id

CSF	*Symbol*	*Prediction*
E	id	E ⇒ TE’’
TE’’	id	T ⇒ VT’’
VT’’E’’	id	V ⇒ id
id T’’E’’	+	T’’ ⇒ ε
id E’’	+	E’’ ⇒ +E
id + E	id	E ⇒ TE’’
id + TE’’	id	T ⇒ VT’’
id + VT’’E’’	id	V ⇒ id
id + id T’’E’’	*	T’’ ⇒ *T
id + id * TE’’	id	T ⇒ VT’’
id + id * VT’’E’’	id	V ⇒ id
id + id * id T’’E’’	-	T’’ ⇒ ε
id + id * id	-	E’’ ⇒ ε
id + id * id	-	-

LL(1) parser

It is a table driven parser for left-to-left parsing

Meaning of LL(1)

L - Scanning of input string from Left to Right
L - Left most derivation
1 - look ahead of one source symbol (the prediction to be made is determined by the next source symbol)

Two functions

First () – the first terminal symbol that is derived from the non terminal

Follow () – the terminal symbol that follows the current non terminal symbol

E ::= TE’
E’ ::= + TE’ | ε
T ::= VT’
T’ ::= * VT’ | ε
V ::= id

Find the first and follow of this grammar.

First and Follow

First (E) = First (T) = First (V) = {id}
First (E’) = {+, ε}
First (T) = First (V) = {id}
First (T’) = {*, ε}
First (V) = {id}

Follow (E) = {$}
Follow (E’) =follow (E) = {$}
Follow (T) = First (E’) = {+,$}
Follow (T’) = {+, $}
Follow (V) = First (T’) = {*,+}

E ::= TE’
E’ ::= + TE’ | ε
T ::= VT’
T’ ::= * VT’ | ε
V ::= id

For creating the LL(1) parser table,

- Place every LHS non-terminal on each row
- Place every terminal on each column
- For very first() of a non-terminal, place the respective production on the appropriate cell
- If the first() contains an ε, then use the follow() of that non-terminal

LL(1) Parsing table

Non-terminal	***Source symbol***			
	id	**+**	*****	**$**
E	E ⇒ TE’			
E’		E’ ⇒ +TE’		E’ ⇒ ε
T	T ⇒ VT’			
T’		T’ ⇒ ε	T’ ⇒ *VT’	T’ ⇒ ε
V	V ⇒ id			

Current sentential form	*Symbol*	*Prediction*
$ E	id	E ⇒ TE’
$ TE’	id	T ⇒ VT’
$ VT’E’	id	V ⇒ id
$ id T’E’	+	T’ ⇒ ε
$ id E’	+	E’ ⇒ +TE’
$ id + TE’	id	T ⇒ VT’
$ id + VT’E’	id	V ⇒ id
$ id + id T’E’	*	T’ ⇒ *VT’
$ id + id *VT’E’	id	V ⇒ id
$ id + id * id T’E’	$	T’ ⇒ ε
$ id + id * id E’	$	E’ ⇒ ε
$ id + id * id	-	-

Exercises

1. Draw the LL(1) parser table.

$S \rightarrow aAC$

$A \rightarrow b$

$C \rightarrow d$

2. Draw the LL(1) parser table.

$A \rightarrow bA \mid cD$

$D \rightarrow aA \mid \varepsilon$

3. Draw the LL(1) parser table.

   $S \rightarrow ABCD$

   $A \rightarrow a \mid \varepsilon$

   $B \rightarrow b \mid \varepsilon$

   $C \rightarrow c$

   $D \rightarrow d \mid \varepsilon$

4. Draw the LL(1) parser table.

   $S \rightarrow Bb \mid Cd$

   $B \rightarrow aB \mid \varepsilon$

   $C \rightarrow cC \mid \varepsilon$

5. Show that the following grammar is not LL(1)

$A \rightarrow d\ A$

$A \rightarrow d\ B$

$A \rightarrow f$

$B \rightarrow g$

6. Find out if the grammar is LL(1) or not?

$S \rightarrow aB \mid \varepsilon$

$B \rightarrow bC \mid \varepsilon$

$C \rightarrow cS \mid \varepsilon$

7. Find out if grammar is LL(1) or not?

$E \rightarrow aEF \mid \varepsilon$

$F \rightarrow d \mid \varepsilon$

8. Find out if grammar is LL(1) or not?

$L \rightarrow X$

$X \rightarrow Yy \mid Za$

$Y \rightarrow cY \mid \varepsilon$

$Z \rightarrow zZ \mid \varepsilon$

9. Show that the following grammar is not LL(1).

$S \rightarrow X\ d$

$X \rightarrow C$

$X \rightarrow B\ a$

$C \rightarrow \varepsilon$

$B \rightarrow d$

10. Given the grammar:

$$S \rightarrow XX$$
$$X \rightarrow xX$$
$$X \rightarrow y$$

(a) Show that it is LL(1).
(b) Create a LL(1) parsing table.
(c) Parse the string *xyyx* .
(d) Draw the parse tree.

Recursive descent parser

Kindly study this topic by yourself