CSCI 742 - Compiler Construction

Lecture 14

Top-Down vs. Bottom-up Parsing

Instructor: Hossein Hojjat

February 24, 2017

Recap: Compiler Phases

Source Code
(concrete syntax)

```
if (x==0) x=x+1;
```

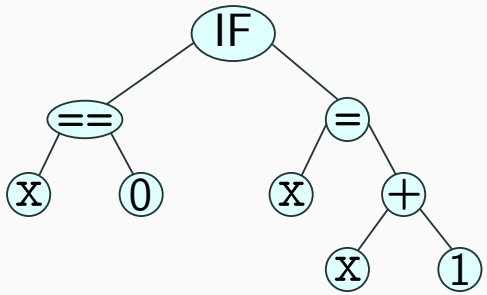
Lexical Analysis

Token Stream

```
if ( x == 0 ) x = x + 1 ;
```

Syntax Analysis
(Parsing)

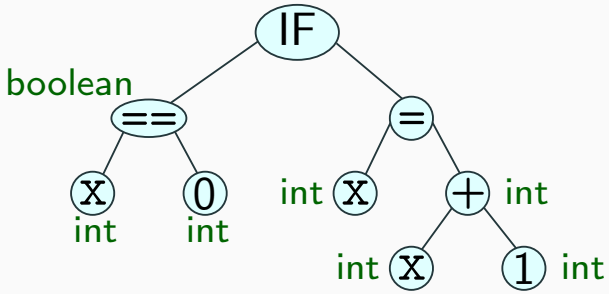
Abstract Syntex Tree
(AST)



Error

Semantic Analysis
(Name Analysis,
Type Analysis, ...)

Attributed AST



Code Generation

Machine Code

```
16: iload_2
17: ifne   24
20: iload_2
21: iconst_1
22: iadd
23: istore_2
24: ...
```

Recap: Leftmost and Rightmost Derivations

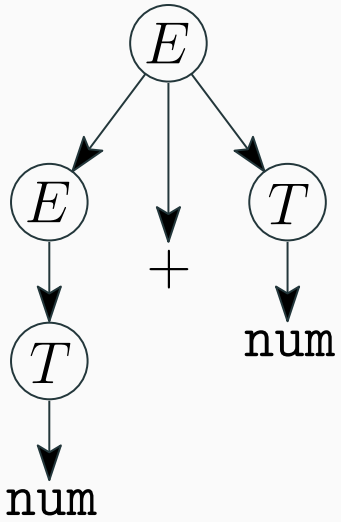
- Grammar for additive arithmetic expressions:

$$E \to E + T$$
$$E \to T$$
$$T \to \texttt{num}$$

- Derivation for $\texttt{num} + \texttt{num}$:

Leftmost Derivation:

$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow T + T \\ &\Rightarrow \texttt{num} + T \\ &\Rightarrow \texttt{num} + \texttt{num} \end{aligned}$$



Rightmost Derivation:

$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow E + \texttt{num} \\ &\Rightarrow T + \texttt{num} \\ &\Rightarrow \texttt{num} + \texttt{num} \end{aligned}$$

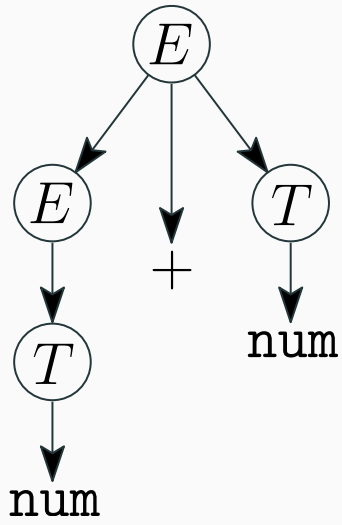
Recap: Approaches to Parsing

Top Down (Goal driven)

- Start from the start non-terminal
- Grow parse tree downwards to match the input word
- Easier to understand and program manually

Bottom Up (Data Driven)

- Start from the input word
- Build up parse tree which has start non-terminal as root
- More powerful and used by most parser generators



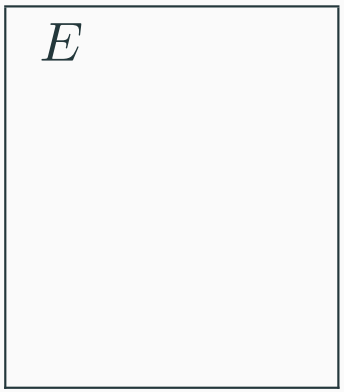
Parsing: Top-down vs. Bottom-up

input: `num + num`

grammar:

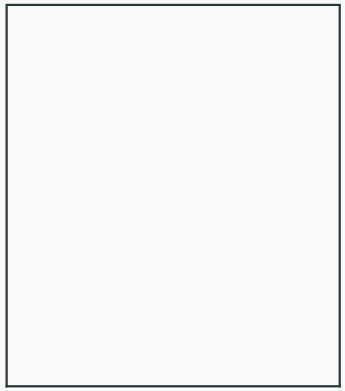
$$E \to E + T$$
$$E \to T$$
$$T \to \texttt{num}$$

Top-down Parsing



Finds leftmost derivation

Bottom-up Parsing



Finds reverse rightmost derivation

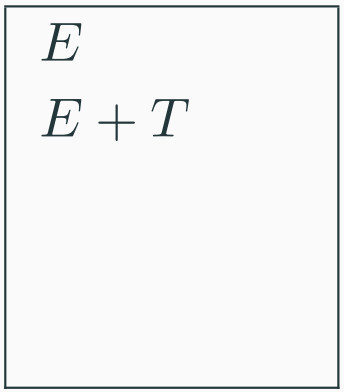
Parsing: Top-down vs. Bottom-up

input: `num + num`

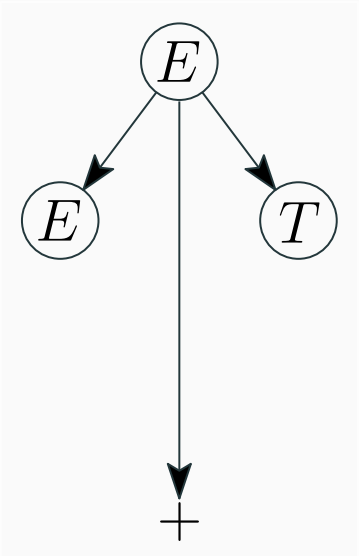
grammar:

$$E \to E + T$$
$$E \to T$$
$$T \to \texttt{num}$$

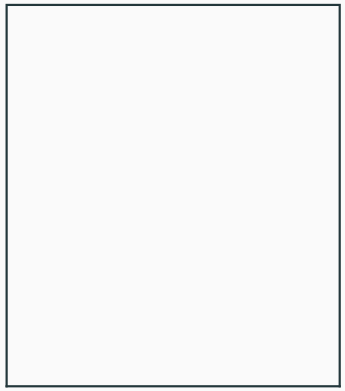
Top-down Parsing



Finds leftmost derivation



Bottom-up Parsing



Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: `num + num`

grammar:

$$E \to E + T$$
$$E \to T$$
$$T \to \texttt{num}$$

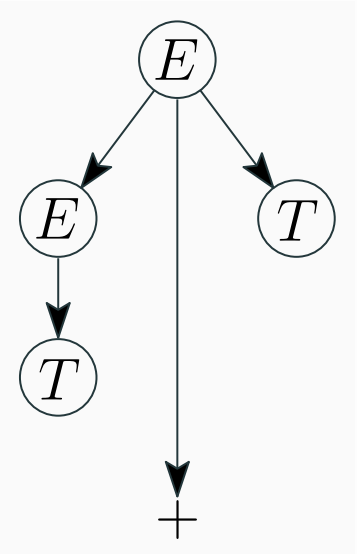
Top-down Parsing

E

$E + T$

$T + T$

Finds leftmost derivation



Bottom-up Parsing

Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: num + num

grammar:

$E \to E + T$

$E \to T$

$T \to \texttt{num}$

Top-down Parsing

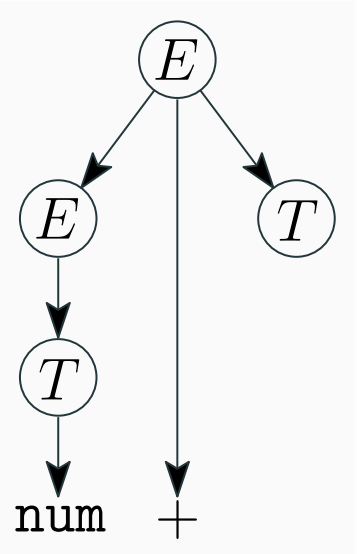
E

$E + T$

$T + T$

$\texttt{num} + T$

Finds leftmost derivation



Bottom-up Parsing

Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: num + num

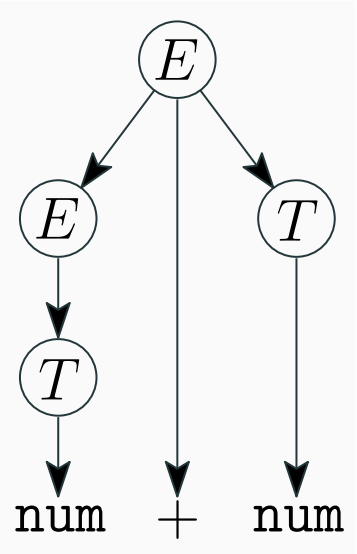
grammar:

$$E \to E + T$$
$$E \to T$$
$$T \to \texttt{num}$$

Top-down Parsing

E
$E + T$
$T + T$
$\texttt{num} + T$
$\texttt{num} + \texttt{num}$

Finds leftmost derivation



Bottom-up Parsing

Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: $\texttt{num} + \texttt{num}$

grammar:

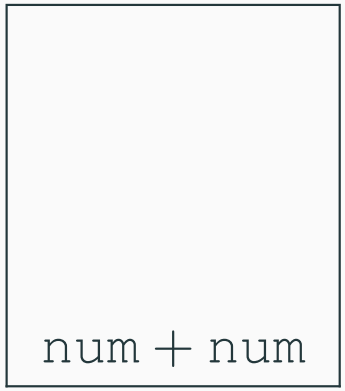
$$E \to E + T$$
$$E \to T$$
$$T \to \texttt{num}$$

Top-down Parsing

E
$E + T$
$T + T$
$\texttt{num} + T$
$\texttt{num} + \texttt{num}$

Finds leftmost derivation

Bottom-up Parsing



Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: num + num

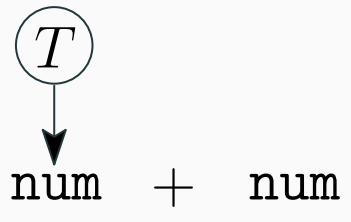
grammar:

$$E \to E + T$$
$$E \to T$$
$$T \to \text{num}$$

Top-down Parsing

E
$E + T$
$T + T$
num $+ T$
num + num

Finds leftmost derivation



Bottom-up Parsing

$T +$ num
num + num

Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: num + num

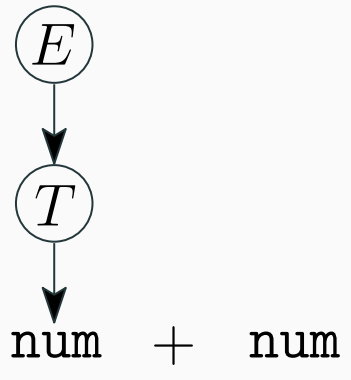
grammar:

$$E \to E + T$$
$$E \to T$$
$$T \to \text{num}$$

Top-down Parsing

E
$E + T$
$T + T$
num $+ T$
num + num

Finds leftmost derivation



Bottom-up Parsing

$E +$ num
$T +$ num
num + num

Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: num + num

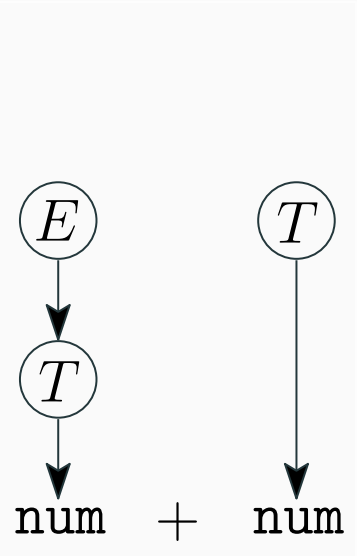
grammar:

$$E \to E + T$$
$$E \to T$$
$$T \to \text{num}$$

Top-down Parsing

E
$E + T$
$T + T$
$\text{num} + T$
$\text{num} + \text{num}$

Finds leftmost derivation



Bottom-up Parsing

$E + T$
$E + \text{num}$
$T + \text{num}$
$\text{num} + \text{num}$

Finds reverse rightmost derivation

Parsing: Top-down vs. Bottom-up

input: num + num

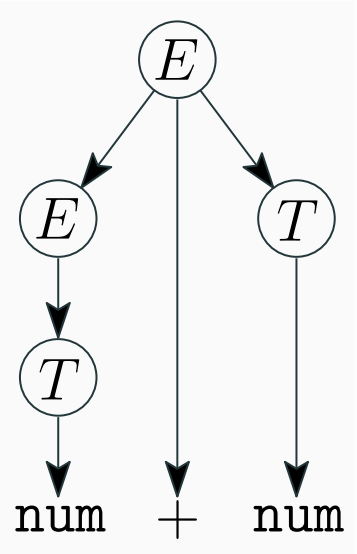
grammar:

$$E \to E + T$$
$$E \to T$$
$$T \to \text{num}$$

Top-down Parsing

E
$E + T$
$T + T$
$\text{num} + T$
$\text{num} + \text{num}$

Finds leftmost derivation



Bottom-up Parsing

E
$E + T$
$E + \text{num}$
$T + \text{num}$
$\text{num} + \text{num}$

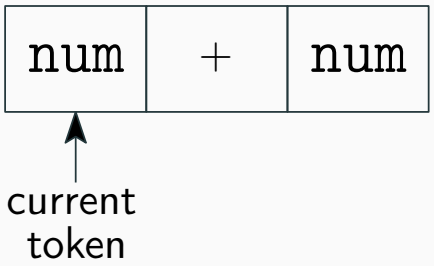
Finds reverse rightmost derivation

Parsing Complexity

- **CYK:** Parsing algorithm for arbitrary context-free grammar
- Worst-case asymptotic complexity for input size n: $O(n^3)$
- For certain classes of constrained CFGs, we can always parse in linear time
  - LL parsers (Use a top-down strategy)
  - LR parsers (Use a bottom-up strategy)
- The first L means the parser reads input from Left to right without backing up
- LL: Left-to-right scan, Leftmost derivation
- LR: Left-to-right scan, Rightmost derivation in reverse

Lookahead Input Symbols

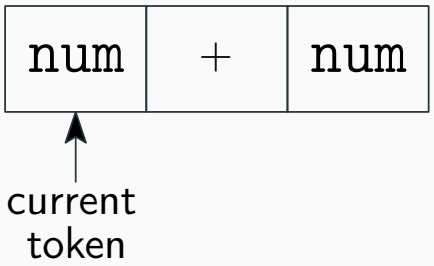
- Build a top-down parse tree for the following input:



1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

Lookahead Input Symbols

- Build a top-down parse tree for the following input:



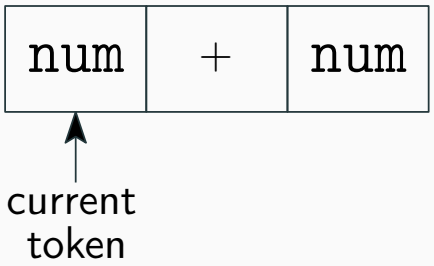
1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

Backtracking:

Make a choice of a production rule, if it fails backtrack and evaluate the next choice

Lookahead Input Symbols

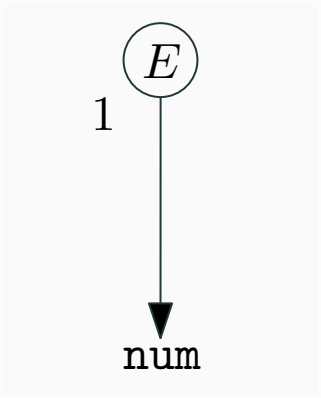
- Build a top-down parse tree for the following input:



1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

Backtracking:

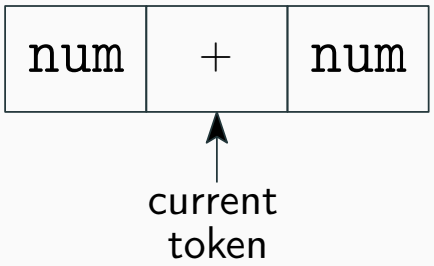
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Matches input token, choice is accepted for now

Lookahead Input Symbols

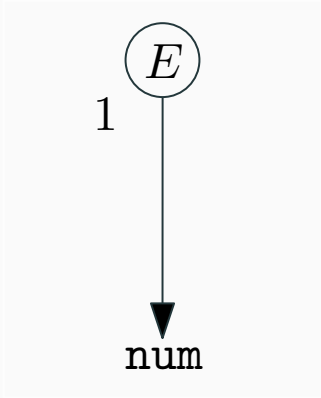
- Build a top-down parse tree for the following input:



1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

Backtracking:

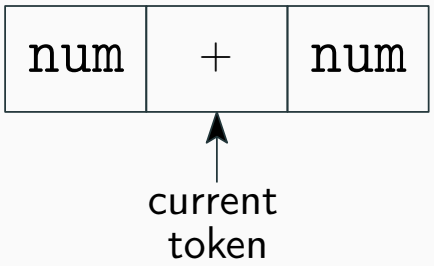
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Matches input token, choice is accepted for now

Lookahead Input Symbols

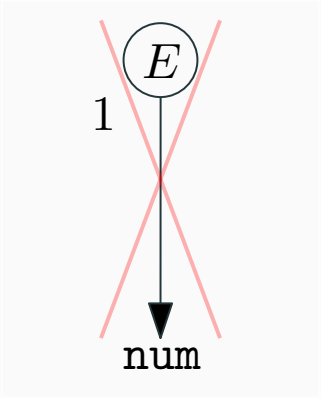
- Build a top-down parse tree for the following input:



1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

Backtracking:

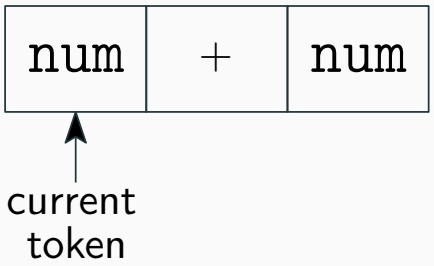
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Can't match input token, need to backtrack

Lookahead Input Symbols

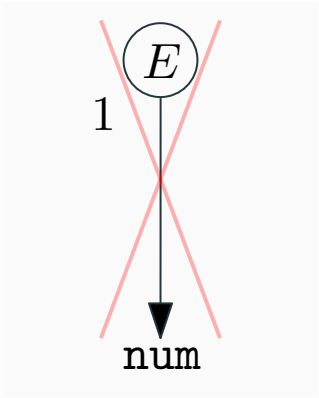
- Build a top-down parse tree for the following input:



1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

Backtracking:

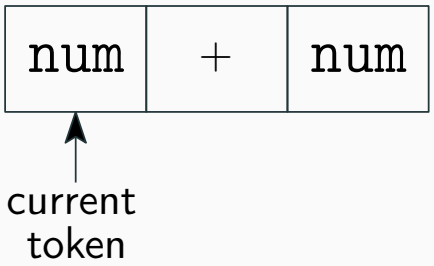
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Can't match input token, need to backtrack

Lookahead Input Symbols

- Build a top-down parse tree for the following input:



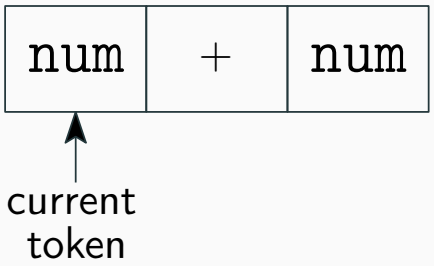
1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

Backtracking:

Make a choice of a production rule, if it fails backtrack and evaluate the next choice

Lookahead Input Symbols

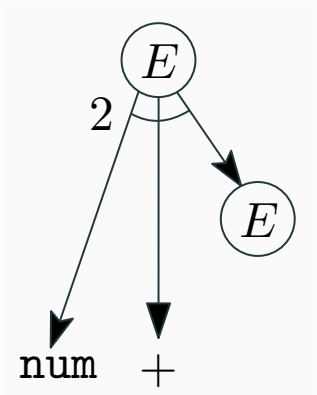
- Build a top-down parse tree for the following input:



1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

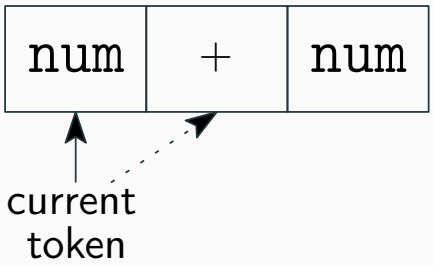
Backtracking:

Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Lookahead Input Symbols

- Build a top-down parse tree for the following input:



1) $E \rightarrow \texttt{num}$
2) $E \rightarrow \texttt{num} + E$

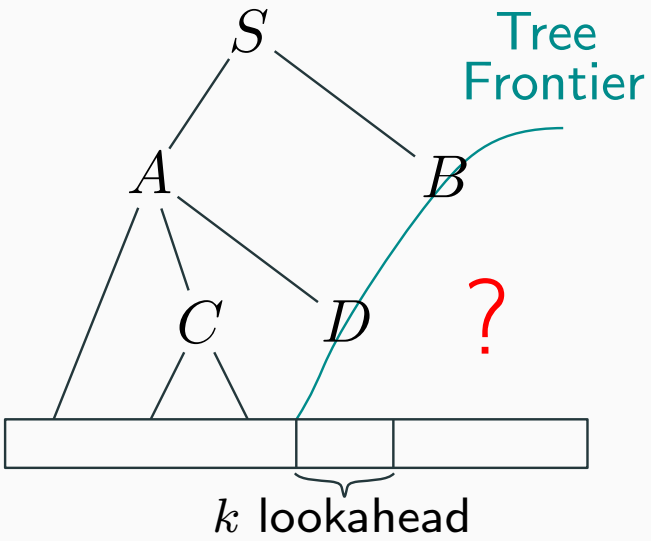
Predictive Parsing:

- Allow parser to "lookahead" k number of tokens from the input
- Decide which production to apply based on next tokens
- Efficient: no need to backtrack
- LL(1): Parser can only look at current token
- LL(2): Parser can only look at current token and the token follows it
- LL(k): Parser can look at k tokens from input

LL(k) Parsing

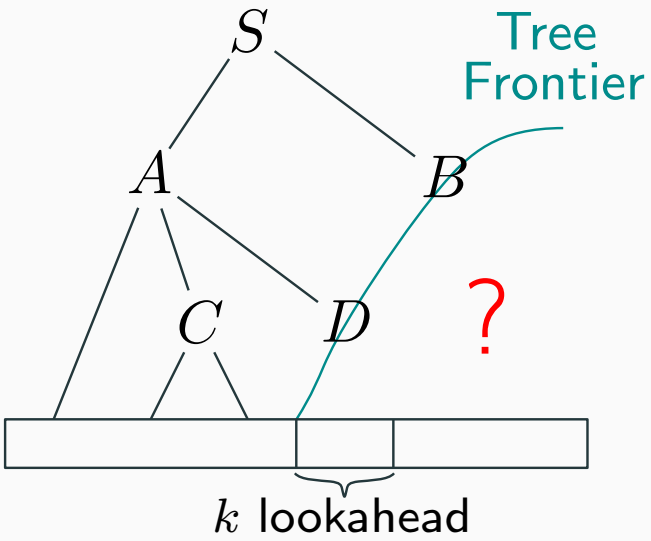
- Determine a leftmost derivation of the input while:
  - Read the input from Left to right
  - Look ahead at most k input tokens
- Starting from the start symbol, grow a parse tree top-down in left-to-right preorder while:
  - Read the input from Left to right
  - Look ahead at most k input tokens beyond the input prefix matched by the parse tree derived so far

LL(k) Parsing



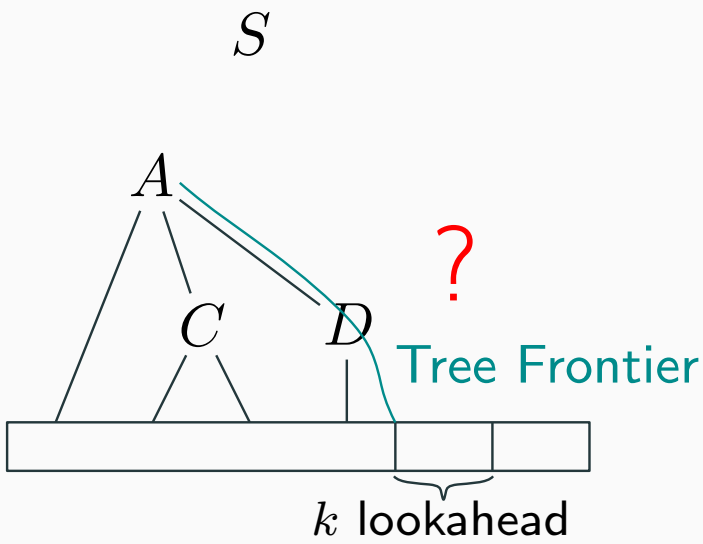
- Parse tree from S to the examined input is complete
- Look-ahead tokens must fully specify the parse tree from S to the input symbol
- In the example we have to know that $S \to AB$ before we even see any of B

LL(k) Parsing



- Assume there are two production rules for D:
  $D \to \alpha_1 \mid \alpha_2 \quad (\alpha_i \in (N \cup T)^*)$
- If $DB \Rightarrow^* w_1$ and $DB \Rightarrow^* w_2$ (w_i is a word)
- If $\alpha_2 \neq \alpha_2$ then w_1 and w_2 must differ in first k symbols

Bottom-up Parsing



- Bottom-up parser builds the tree only above the examined input
- Although we are at the same point in the input string, the production $S \to AB$ has not been specified yet
- This delayed decision allows us to parse more grammars than predictive top-down parsing (LL)

Exercise

Question

Is the following grammar LL(k)? If yes, for which value of k?

$$S \to AB$$
$$A \to aAb \mid \epsilon$$
$$B \to bB \mid \epsilon$$

Exercise

Question

Is the following grammar LL(k)? If yes, for which value of k?

$$S \to AB$$
$$A \to aAb \mid \epsilon$$
$$B \to bB \mid \epsilon$$

Answer

Grammar is LL(1).

Any derivation starts with $S \Rightarrow AB$.

The next derivation step uses one of the productions $A \to aAb$ or $A \to \epsilon$ based on the next current token.

The same argument holds for B-productions.

Exercise

Question

Is the following grammar LL(k)? If yes, for which value of k?

$$S \to Ab \mid Ac$$
$$A \to aA \mid \epsilon$$

Exercise

Question

Is the following grammar LL(k)? If yes, for which value of k?

$$S \to Ab \mid Ac$$
$$A \to aA \mid \epsilon$$

Answer

- Grammar is not LL(k) parser for any finite k
- Expanding S to one of the alternatives is the first step a top down parser has to do
- There can always be a word that needs more than k lookahead
- For a word beginning with k a's parser needs to look at at least $(k+1)$ lookahead tokens to make the decision

Left-recursive Grammars

- Left recursive grammars cannot be parsed by a LL(k)-parser
- Predictive parser uses the lookahead tokens to choose the correct production rule
- For each k lookahead tokens there must be a unique production
- On a left-recursive grammar the algorithm may try to expand a production without consuming any input
- Parse tree continuously get expanded without any advance in input
- Parsing process may never terminate!