CSCI 742 - Compiler Construction

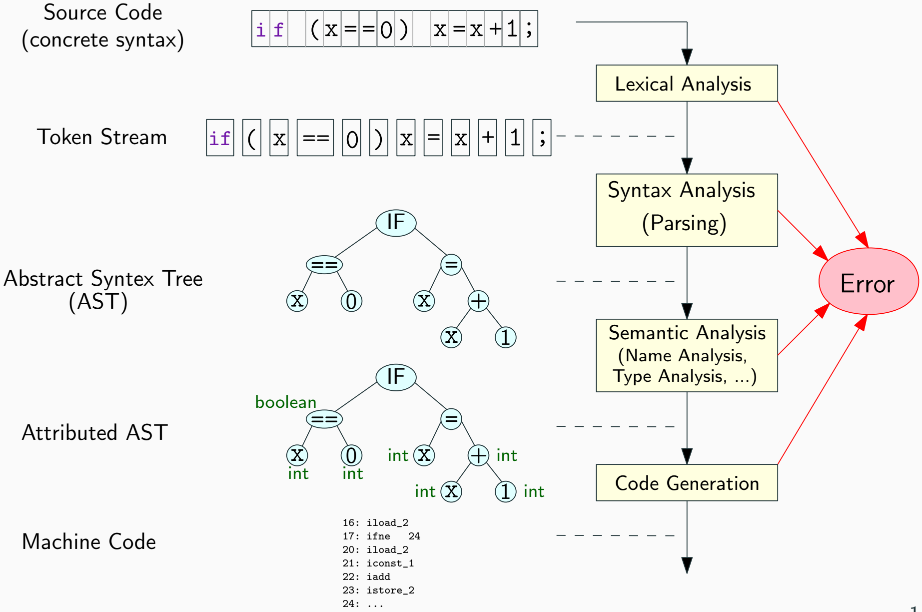
Lecture 10

Top-Down vs. Bottom-up Parsing

Instructor: Hossein Hojjat

February 7, 2018

Recap: Compiler Phases



Source Code
(concrete syntax)

Token Stream

Abstract Syntex Tree
(AST)

Attributed AST

Machine Code

Lexical Analysis

Syntax Analysis
(Parsing)

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

Code Generation

Error

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

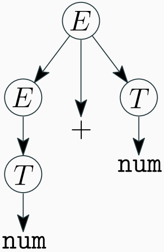
Approaches to Parsing

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



Directionality

Directional Methods

- Process the input symbol by symbol from Left to right
- Advantage: parsing starts and makes progress before the last symbol of the input is seen
- Example: LL and LR parsers

Non-directional Methods

- Allow access to input in an arbitrary order
- Require the entire input to be in memory before parsing can start
- Advantage: allow more flexible grammars than directional parsers
- Example: CYK parser

Directionality

Directional Methods

- Process the input symbol by symbol from Left to right
- Advantage: parsing starts and makes progress before the last symbol of the input is seen
- Example: LL and LR parsers

Non-directional Methods

- Allow access to input in an arbitrary order
- Require the entire input to be in memory before parsing can start
- Advantage: allow more flexible grammars than directional parsers
- Example: CYK parser

We first focus on directional parsers (will discuss CYK after LL and LR)

Parsing: Top-down vs. Bottom-up (Directional)

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

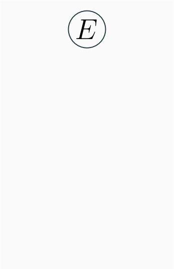
grammar:

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

Top-down Parsing

E

Finds leftmost derivation



Remaining Input: $\texttt{num} \quad + \quad \texttt{num}$

Parsing: Top-down vs. Bottom-up (Directional)

input: `num` + `num`

grammar:

$$E \to E + T$$

$$E \to T$$

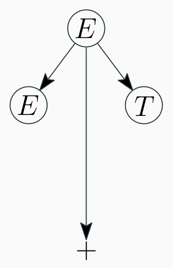
$$T \to \texttt{num}$$

Top-down Parsing

E

$E + T$

Finds leftmost derivation



Remaining Input: `num` + `num`

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

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

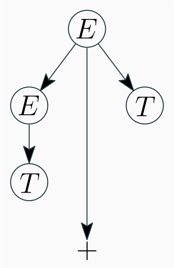
Top-down Parsing

E

$E + T$

$T + T$

Finds leftmost derivation



Remaining Input: $\texttt{num} \ + \ \texttt{num}$

Parsing: Top-down vs. Bottom-up (Directional)

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

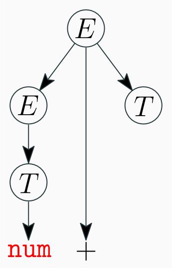
grammar:

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

Top-down Parsing

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

Finds leftmost derivation



Remaining Input: **num** $+$ num

Match Input Token!

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

$$E \to E + T$$

$$E \to T$$

$$T \to \mathrm{num}$$

Top-down Parsing

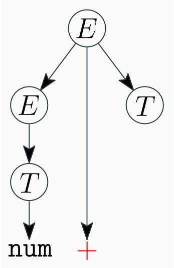
E

$E + T$

$T + T$

$\mathrm{num} + T$

Finds leftmost derivation



Remaining Input: $+ \quad \mathrm{num}$

Match Input Token!

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

$$E \to E + T$$

$$E \to T$$

$$T \to \mathrm{num}$$

Top-down Parsing

E

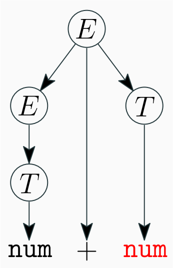
$E + T$

$T + T$

$\mathrm{num} + T$

$\mathrm{num} + \mathrm{num}$

Finds leftmost derivation



Remaining Input: `num`

Match Input Token!

Parsing: Top-down vs. Bottom-up (Directional)

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

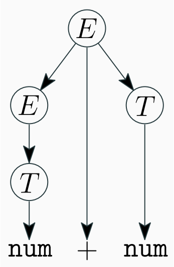
grammar:

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

Top-down Parsing

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

Finds leftmost derivation



Remaining Input:

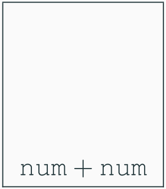
Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

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

Bottom-up Parsing



$\texttt{num} + \texttt{num}$

Finds reverse rightmost derivation

Remaining Input: $\texttt{num} \quad + \quad \texttt{num}$

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

$$E \to E + T$$

$$E \to T$$

$$T \to \mathrm{num}$$

`num`

Bottom-up Parsing

$\mathrm{num} + \mathrm{num}$

Finds reverse rightmost derivation

Remaining Input: $+\ \mathrm{num}$

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:
$$E \to E + T$$
$$E \to T$$
$$T \to \mathrm{num}$$

Bottom-up Parsing

$T + \mathrm{num}$

$\mathrm{num} + \mathrm{num}$

Finds reverse rightmost derivation

Remaining Input: $+ \quad \mathrm{num}$

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow \mathrm{num}$$

Bottom-up Parsing

$E + \mathrm{num}$

$T + \mathrm{num}$

$\mathrm{num} + \mathrm{num}$

Finds reverse rightmost derivation

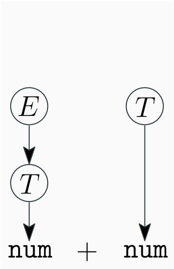
Remaining Input: $+ \;\; \mathrm{num}$

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

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



Bottom-up Parsing

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

Finds reverse rightmost derivation

Remaining Input:

Parsing: Top-down vs. Bottom-up (Directional)

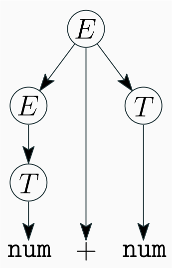
input: $\mathrm{num} + \mathrm{num}$

grammar:

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow \mathrm{num}$$



Bottom-up Parsing

E

$E + T$

$E + \mathrm{num}$

$T + \mathrm{num}$

$\mathrm{num} + \mathrm{num}$

Finds reverse rightmost derivation

Remaining Input:

Parsing: Top-down vs. Bottom-up (Directional)

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

grammar:

$$E \to E + T$$

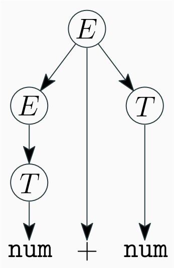
$$E \to T$$

$$T \to \mathrm{num}$$

Top-down Parsing

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

Finds leftmost derivation



Bottom-up Parsing

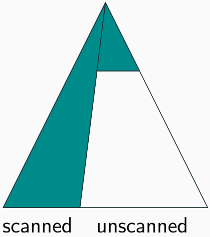
E
$E + T$
$E + \mathrm{num}$
$T + \mathrm{num}$
$\mathrm{num} + \mathrm{num}$

Finds reverse rightmost derivation

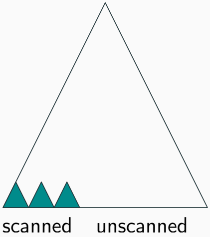
Parsing: Top-down vs. Bottom-up (Directional)

Bottom-up: Don't need to figure out as much of the parse tree for a given amount of input (more powerful)

Top-down: Easier to understand and program manually



Top-down



Bottom-up

Parsing Complexity

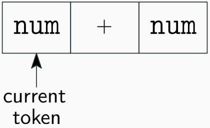
- For certain classes of constrained CFGs, we can always parse in **linear** time
  - LL parsers (Use a top-down strategy)
  - LR parsers (Use a bottom-up strategy)
- The first L means the parser reads input from Left to right without backing up

- LL: Left-to-right scan, Leftmost derivation
- LR: Left-to-right scan, Rightmost derivation in reverse

- Any ambiguous CFG can neither be LL nor LR
- **Deterministic:** they produce a single correct parse without guessing or backtracking

Lookahead Input Symbols

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



num	+	num

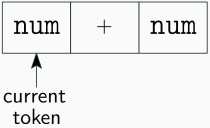
current token

1) $E \rightarrow \mathrm{num}$

2) $E \rightarrow \mathrm{num} + E$

Lookahead Input Symbols

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



1) $E \rightarrow \mathrm{num}$

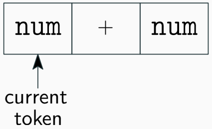
2) $E \rightarrow \mathrm{num} + E$

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:

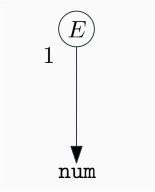


1) $E \rightarrow \mathrm{num}$

2) $E \rightarrow \mathrm{num} + E$

Backtracking:

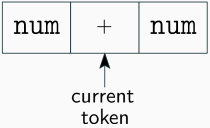
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Matches input token, choice is accepted for now

Lookahead Input Symbols

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

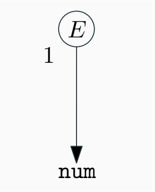


1) $E \rightarrow \text{num}$

2) $E \rightarrow \text{num} + E$

Backtracking:

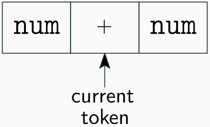
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Matches input token, choice is accepted for now

Lookahead Input Symbols

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

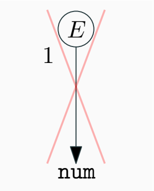


1) $E \rightarrow \mathrm{num}$

2) $E \rightarrow \mathrm{num} + E$

Backtracking:

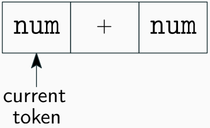
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Can't match input token, need to backtrack

Lookahead Input Symbols

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

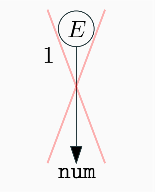


1) $E \to \text{num}$

2) $E \to \text{num} + E$

Backtracking:

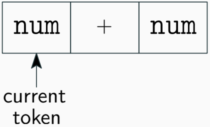
Make a choice of a production rule, if it fails backtrack and evaluate the next choice



Can't match input token, need to backtrack

Lookahead Input Symbols

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



num	+	num

current token

1) $E \rightarrow \mathrm{num}$

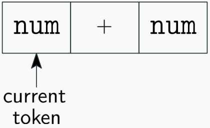
2) $E \rightarrow \mathrm{num} + E$

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:

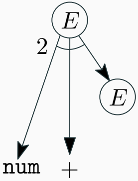


1) $E \rightarrow \mathrm{num}$

2) $E \rightarrow \mathrm{num} + E$

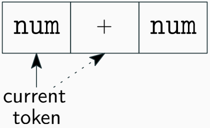
Backtracking:

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) $E \rightarrow \mathrm{num}$

2) $E \rightarrow \mathrm{num} + E$

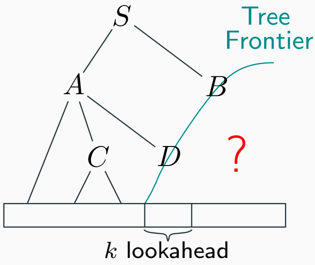
Predictive Parsing:

- 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

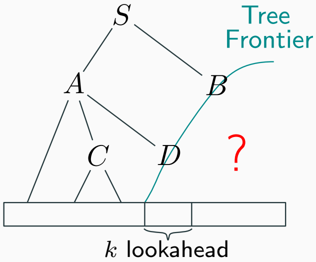
- Determine a leftmost derivation of the input while:
  - Read the input from Left to right
  - Look ahead at most k input tokens
- Starting from the start symbol, grow a parse tree top-down in left-to-right pre-order while:
  - Read the input from Left to right
  - Look ahead at most k input tokens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

LL(k) Parsing



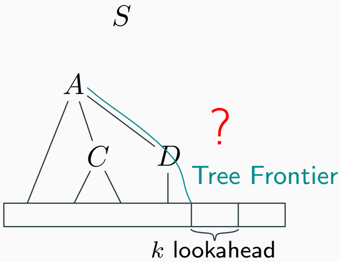
- Parse tree from S to the examined input is complete
- Look-ahead tokens must fully specify the parse tree from S to the input symbol
- In the example we have to know that $S \rightarrow AB$ before we even see any of B

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 $S \rightarrow AB$ 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?

$$\begin{aligned} S &\to AB \\ A &\to aAb \mid \epsilon \\ B &\to bB \mid \epsilon \end{aligned}$$

Exercise

Question

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

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

Answer

Grammar is LL(1).

Any derivation starts with $S \Rightarrow AB$.

The next derivation step uses one of the productions $A \rightarrow aAb$ or $A \rightarrow \epsilon$ 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?

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow aaA \mid aa \\ B &\rightarrow aaB \mid a \end{aligned}$$

Exercise

Question

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

$$\begin{aligned} S &\to A \mid B \\ A &\to a \mid c \\ B &\to b \mid c \end{aligned}$$

Exercise

Question

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

$$
\begin{aligned}
S &\rightarrow aaA \mid AB \\
A &\rightarrow a \mid \epsilon \mid ab \\
B &\rightarrow b
\end{aligned}
$$

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!