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 Syntax Tree (AST)

```
IF
==
  x
  0
  =
  x
  +
  x
  1
```

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

```
boolean
==
  x
  0
  int
  int
  int
  +
```

Attributed AST

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

Code Generation

Machine Code
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
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: num + num

Top-down Parsing

\[ E \Rightarrow E + T \]
\[ E \rightarrow T \]
\[ T \rightarrow \text{num} \]

Finds leftmost derivation

Remaining Input: num + num
Parsing: Top-down vs. Bottom-up (Directional)

**input:** num + num  

**grammar:**

\[
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow \text{num}
\]

Top-down Parsing

```
E
E + T
```

Finds leftmost derivation

Remaining Input: num + num
Parsing: Top-down vs. Bottom-up (Directional)

**input:** num + num

**grammar:**

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

Top-down Parsing

```
E
E + T
T + T
```

Finds leftmost derivation

Remaining Input: num + num
Parsing: Top-down vs. Bottom-up (Directional)

**input:** \( \text{num} + \text{num} \)

**grammar:**

- \( E \rightarrow E + T \)
- \( E \rightarrow T \)
- \( T \rightarrow \text{num} \)

**Top-down Parsing**

- \( E \)
- \( E + T \)
- \( T + T \)
- \( \text{num} + T \)

Finds leftmost derivation

**Remaining Input:** \( \text{num} + \text{num} \)

Match Input Token!
input: num + num

grammar:
- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow \text{num}$

Top-down Parsing

Finds leftmost derivation

Remaining Input: + num

Match Input Token!
Parsing: Top-down vs. Bottom-up (Directional)

**input:** num + num

**grammar:**

\[
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow \text{num}
\]

**Top-down Parsing**

- \( E \)
- \( E + T \)
- \( T + T \)
- \( \text{num} + T \)
- \( \text{num} + \text{num} \)

Finds leftmost derivation

**Remaining Input:** num

Match Input Token!
Parsing: Top-down vs. Bottom-up (Directional)

**input:** num + num  

**grammar:**

\[
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow \text{num}
\]

**Top-down Parsing**

\[
\begin{array}{l}
E \\
E + T \\
T + T \\
\text{num} + T \\
\text{num} + \text{num}
\end{array}
\]

Finds leftmost derivation

**Remaining Input:**

\[
\begin{array}{c}
\text{num} \\
+ \\
\text{num}
\end{array}
\]
**Parsing: Top-down vs. Bottom-up (Directional)**

**Input:** `num + num`  

**Grammar:**

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

**Bottom-up Parsing**

- Finds reverse rightmost derivation

**Remaining Input:** `num + num`
**Parsing: Top-down vs. Bottom-up (Directional)**

**input:** $num + num$

**grammar:**
- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow num$

**Bottom-up Parsing**

- Finds reverse rightmost derivation
- Remaining Input: $+ num$

**Match Input Token!**

4
**Parsing: Top-down vs. Bottom-up (Directional)**

**input:** \( \text{num} + \text{num} \)

**grammar:**

\[
\begin{align*}
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow \text{num}
\end{align*}
\]

**Bottom-up Parsing**

```
+ num
```

Finds reverse rightmost derivation

**Remaining Input:**

\( + \, \text{num} \)
**Parsing: Top-down vs. Bottom-up (Directional)**

**input:** `num + num`

**grammar:**

\[
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow num
\]

**Bottom-up Parsing**

Finds reverse rightmost derivation

Remaining Input: `+ num`
input: \( \text{num} + \text{num} \)

grammar:

\[

e \rightarrow e + t \\
 e \rightarrow t \\
 t \rightarrow \text{num}
\]

Bottom-up Parsing

\[

e + t \\
 e + \text{num} \\
 t + \text{num} \\
 \text{num} + \text{num}
\]

Finds reverse rightmost derivation

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

**input:** num + num

**grammar:**

\[
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow \text{num}
\]

**Bottom-up Parsing**

\[
\begin{align*}
E & \\
E + T & \\
E + \text{num} & \\
T + \text{num} & \\
\text{num + num} & \\
\end{align*}
\]

Finds reverse rightmost derivation

**Remaining Input:**

E E T
+
T
num num
Parsing: Top-down vs. Bottom-up (Directional)

**input:** num + num

**grammar:**
- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow \text{num}$

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

```
num + num
```

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

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

```
num + num
```

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.
Lookahead Input Symbols

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

```
num  +  num
```

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

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

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

Matches input token, choice is accepted for now
• Build a **top-down** parse tree for the following input:

```
num | + | num
```

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:

```
num + num
```

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

```
1) E → num
2) E → num + E
```

Can’t match input token, need to backtrack
Lookahead Input Symbols

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

```
num + num
```

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

Can’t match input token, need to backtrack
Lookahead Input Symbols

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

```
num  +  num
```

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.
Lookahead Input Symbols

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

```
num + num
```

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

1) \( E \rightarrow \text{num} \)
2) \( E \rightarrow \text{num} + E \)
• Build a **top-down** parse tree for the following input:

```
  num | + | num
```  

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

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

**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
• 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 $B$
• Assume there are two production rules for $D$:
  $$D \rightarrow \alpha_1 \mid \alpha_2 \quad (\alpha_i \in (N \cup T)^*)$$

• 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 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 $k$?

\[
S \rightarrow AB \\
A \rightarrow aAb \mid \epsilon \\
B \rightarrow bB \mid \epsilon
\]

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

\[
S \rightarrow A \mid B \\
A \rightarrow aaA \mid aa \\
B \rightarrow aaB \mid a
\]
Exercise

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

\[
S \rightarrow A \mid B \\
A \rightarrow a \mid c \\
B \rightarrow b \mid c
\]
Question

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

\[
S \rightarrow aaA \mid AB
\]
\[
A \rightarrow a \mid \epsilon \mid ab
\]
\[
B \rightarrow b
\]
Question

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

$$S \rightarrow Ab \mid Ac$$
$$A \rightarrow aA \mid \epsilon$$
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!