50.051 Programming Language Concepts

W11-S2 Bottom-Up Parsing (Part 1)

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Different types of Parsing

Two big families of parsing algorithms

Top-Down Parsing

- Seen before, best we could do was in the case of LL(1) grammars, but very few grammars are going to be LL(k).

Bottom-Up Parsing

- Start from the input string *x*, whose syntax neds to be verified.
- Work you way back to a start symbol.
- Basically, the Top-Down parsing task, but in reverse!
- Builds on ideas of Top-Down parsing, but more efficient, and will work on non-LL(k) grammars.

Introduction example for Bottom-Up Parsing

Fact #1 (to be confirmed later): Bottom-Up Parsers can deal with non-LL(k) grammars and can also handle left-recursive grammars.

- Consider the following grammar:

$$E \rightarrow E + (E) \mid int$$

- This CFG is this not LL(1).
- If not convinced, consider the string int + (int) + (int).

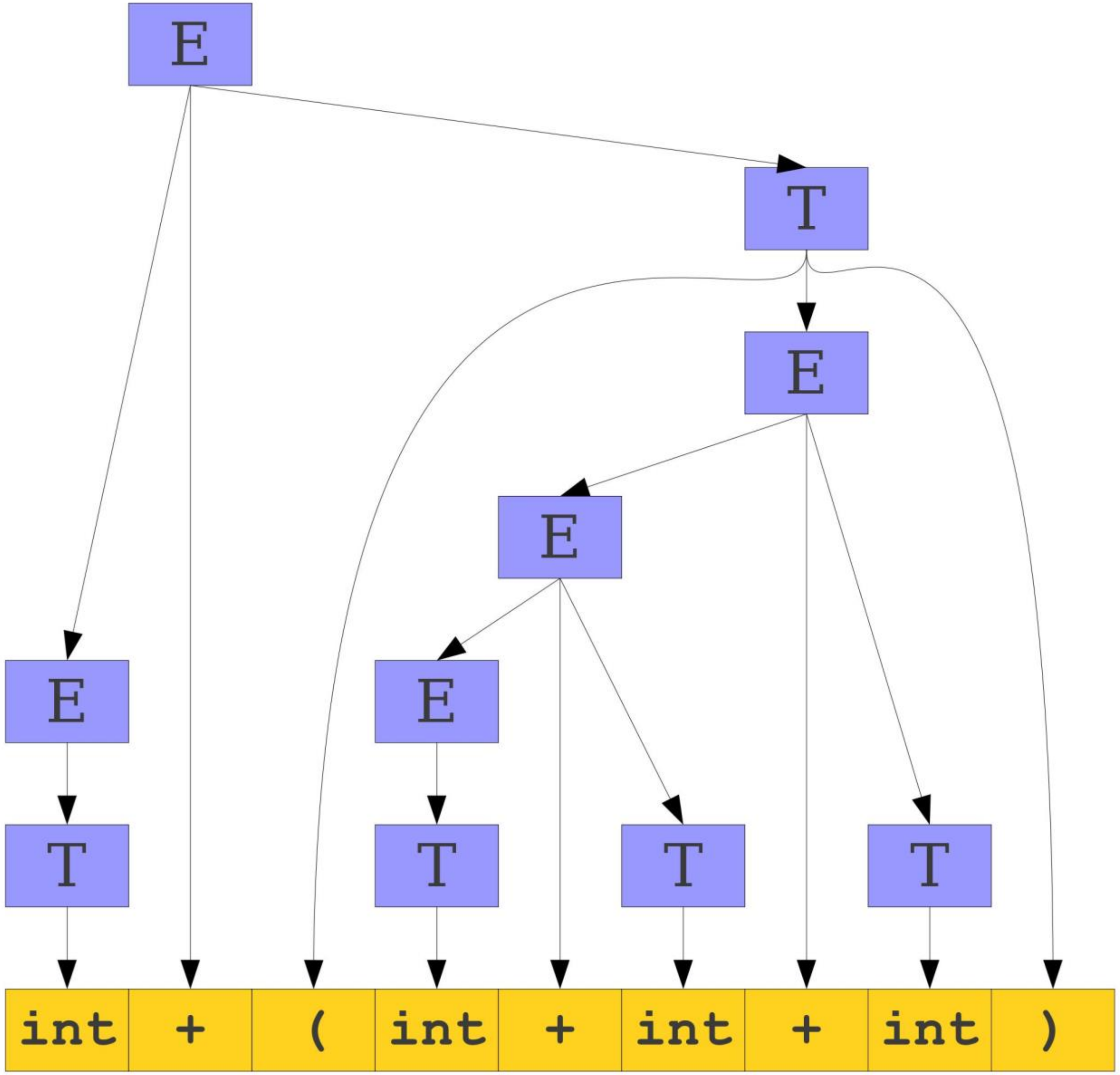
Introduction example for Bottom-Up Parsing

Procedure, in Layman terms: Bottom-Up Parsers attempts to reduce a string *x* to the start symbol by inverting productions.

- At the beginning, *x* should consist only of terminal symbols.
- Identify a substring β in *x*, such that $A \to \beta$ is a production rule of our CFG. In other words, it means $x = \alpha\beta\gamma$, with α and γ being strings of some sort (could be empty strings).
- Replace β with A inside of *x*, replacing *x* with $\alpha A\gamma$.
- Keep on doing so, until *x* becomes the start symbol S of the CFG.

Introduction example for Bottom-Up Parsing

$$E \rightarrow T$$
$$E \rightarrow E + T$$
$$T \rightarrow \texttt{int}$$
$$T \rightarrow (E)$$



Introduction example for Bottom-Up Parsing

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow \texttt{(}E\texttt{)}$

$$
\begin{aligned}
& \texttt{int + (int + int + int)} \\
\Rightarrow\ & T \texttt{ + (int + int + int)} \\
\Rightarrow\ & E \texttt{ + (int + int + int)} \\
\Rightarrow\ & E \texttt{ + (}T\texttt{ + int + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + int + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + }T\texttt{ + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + }T\texttt{)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{)} \\
\Rightarrow\ & E \texttt{ + } T \\
\Rightarrow\ & E
\end{aligned}
$$

Introduction example for Bottom-Up Parsing

$$
\begin{aligned}
E &\rightarrow T \\
E &\rightarrow E + T \\
T &\rightarrow \texttt{int} \\
T &\rightarrow (E)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{int + (int + int + int)} \\
\Rightarrow\ & T \texttt{ + (int + int + int)} \\
\Rightarrow\ & E \texttt{ + (int + int + int)} \\
\Rightarrow\ & E \texttt{ + (}T\texttt{ + int + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + int + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + }T\texttt{ + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + int)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{ + }T\texttt{)} \\
\Rightarrow\ & E \texttt{ + (}E\texttt{)} \\
\Rightarrow\ & E \texttt{ + } T \\
\Rightarrow\ & E
\end{aligned}
$$

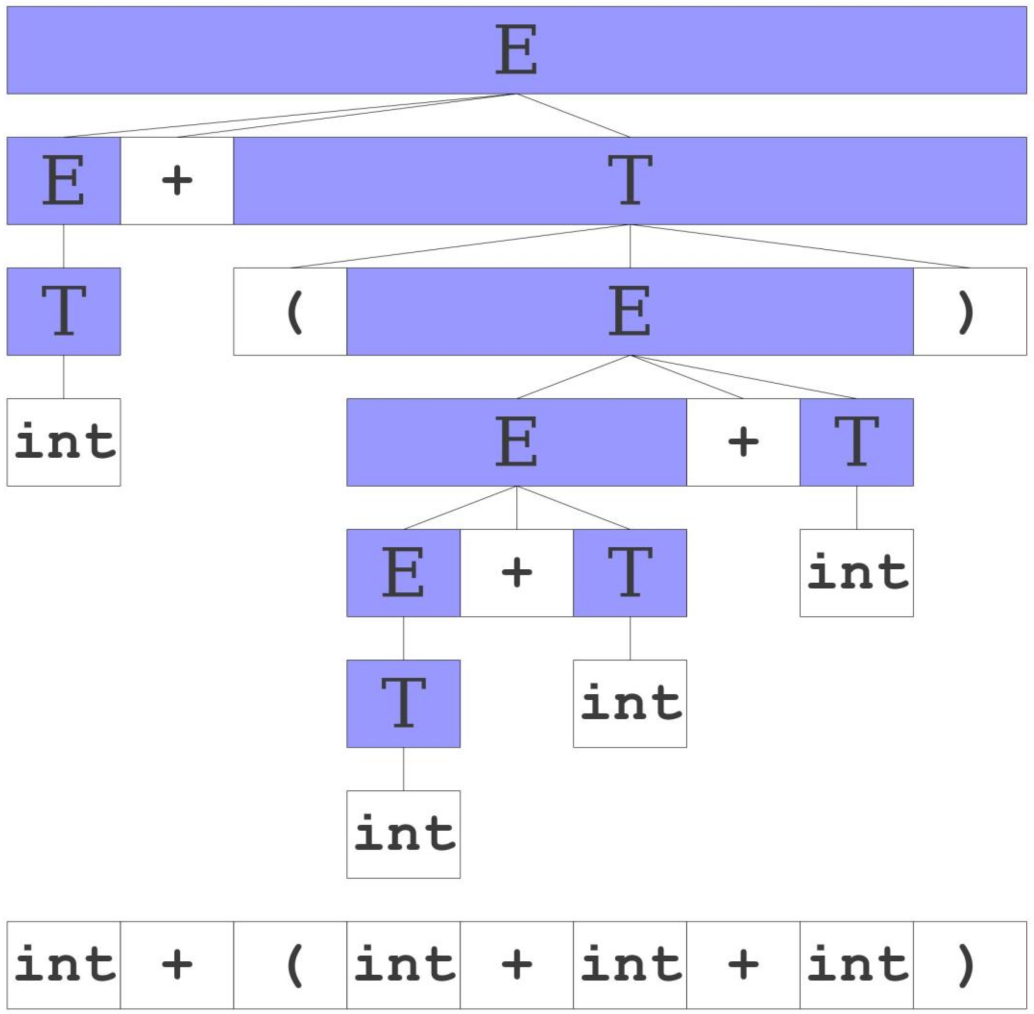
Introduction example for Bottom-Up Parsing

Fact #2: A Bottom-Up parser traces a rightmost derivation in reverse.

- This has an interesting consequence…
- Let $\alpha\beta\gamma$ be a step of a bottom-up parse.
- Assume the next reduction is by $A \rightarrow \beta$
- **Then γ is necessarily a string of terminals!**
- Why? Because $\alpha A\gamma \rightarrow \alpha\beta\gamma$ is a step in a rightmost derivation!
  (If γ does not change, it means it does not contain any non-terminals)

Introduction example for Bottom-Up Parsing

int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Introduction example for Bottom-Up Parsing

Follow-up idea from Fact #2: Split the string into two substrings.

- The right substring should consist only of terminal symbols, and has yet to be examined by the parser.
- The left substring could have terminals and non-terminals.
- Mark the dividing point using the | symbol.
  (Note that this symbol | is only for visualization purposes, it is not part of the string to be analysed!)
- At the beginning, the string x is therefore written as *x* = $|x_1 x_2 \ldots x_n\$$, with $x_1, x_2, \ldots x_n$ being terminal symbols.

Shift-Reduce Parsing

Fact #3: A Bottom-up Parser will attempt to revert the string by using only two possible actions.

- **Shifting:** Moves the separator one step to the right (one full symbol).

For instance, E | + (int) becomes E + | (int) after shifting.

- **Reducing:** Apply an inverse production rule at the right of the left end string.

For instance, the string E + (E + (E) |) can be reduced into E + (E |) by using the production E → E + (E).

Practice 1: Shift-Reduce tryout

Question: For the CFG on the right, what is the correct sequence of Reduce and Shift operations to use on the string x below?

int + (int + int + int)

To assist you, we show the derivation on the right, again.

$$
\begin{aligned}
E &\rightarrow T \\
E &\rightarrow E + T \\
T &\rightarrow \texttt{int} \\
T &\rightarrow (E)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{int + (int + int + int)} \\
\Rightarrow\ & T\ \texttt{+ (int + int + int)} \\
\Rightarrow\ & E\ \texttt{+ (int + int + int)} \\
\Rightarrow\ & E\ \texttt{+ (}T\ \texttt{+ int + int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ int + int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ }T\ \texttt{+ int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ }T\texttt{)} \\
\Rightarrow\ & E\ \texttt{+ (}E\texttt{)} \\
\Rightarrow\ & E\ \texttt{+ }T \\
\Rightarrow\ & E
\end{aligned}
$$

Practice 1: Shift-Reduce tryout

Question: For the CFG on the right, what is the correct sequence of Reduce and Shift operations to use on the string x below?

int + (int + int + int)

$$E \rightarrow T$$
$$E \rightarrow E + T$$
$$T \rightarrow \texttt{int}$$
$$T \rightarrow (E)$$

$$
\begin{aligned}
& \texttt{int + (int + int + int)} \\
\Rightarrow\ & T\ \texttt{+ (int + int + int)} \\
\Rightarrow\ & E\ \texttt{+ (int + int + int)} \\
\Rightarrow\ & E\ \texttt{+ (}T\ \texttt{+ int + int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ int + int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ }T\ \texttt{+ int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ int)} \\
\Rightarrow\ & E\ \texttt{+ (}E\ \texttt{+ }T\texttt{)} \\
\Rightarrow\ & E\ \texttt{+ (}E\texttt{)} \\
\Rightarrow\ & E\ \texttt{+ }T \\
\Rightarrow\ & E
\end{aligned}
$$

To assist you, we show the derivation on the right, again.

Answer: Will be shown on board.

Practical implementation of Shift-Reduce

Practical implementation for the Shift-Reduce parser can be done as follows.

- **Left substring** can be implemented by a **stack**.
- Top of the stack is denoted by the separator symbol | .
- **Shifting pushes a terminal symbol on the stack.**
- **Reducing using a production rule A → B pops all the symbols in B off the top of the stack and then pushes a non-terminal symbol A on the stack.**

Practical implementation of Shift-Reduce

Practical implementation for the Shift-Reduce parser can be done as follows.

- **Left substring** can be implemented by a **stack**.
- Top of the stack is denoted by the separator symbol |.
- **Shifting pushes a terminal symbol on the stack.**
- **Reducing using a production rule A → B pops all the symbols in B off the top of the stack and then pushes a non-terminal symbol A on the stack.**
- **Million dollar question: How to decide when to Shift, when to Reduce, and with which production rule from the CFG?**