COMP3173 Compiler Construction
Bottom-Up Parsing

Dr. Zhiyuan Li

Outline

- Limit of Top-Down Parsing
- Bottom-Up Parsing
- LR Parsing Algorithm

Limit of Top-Down Parsing

- Back to the example in Lecture 5. The following grammar is left-recursive.

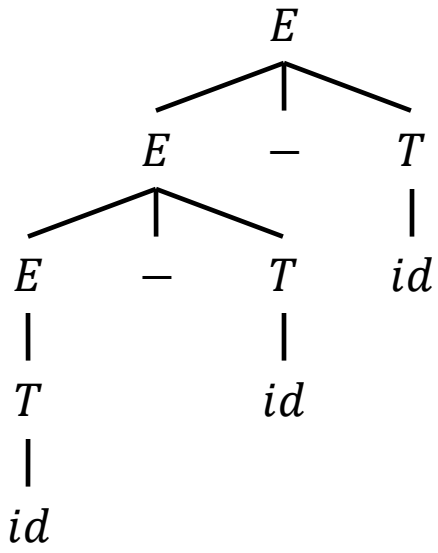
$$\begin{aligned} E &\rightarrow E - T | T \\ T &\rightarrow id \end{aligned}$$

- Thus, top-down parsing needs to eliminate left recursion first.

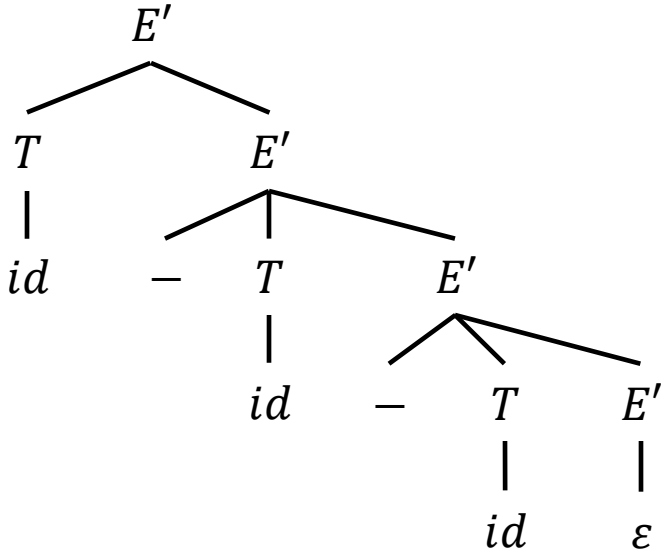
$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow -TE' | \varepsilon \\ T &\rightarrow id \end{aligned}$$

Limit of Top-Down Parsing

- However, the conversion changes the structure of some sentences, like $id - id - id$.



$(id - id) - id$



$id - (id - id)$

Bottom-Up Parsing

- Bottom-up parsing is more natural to human.
- Think about how we analyze the arithmetic expressions. First, we calculate the subexpression in parentheses or the operator of the highest precedence.
- Then, the intermediate result will involve in the following calculations. The procedure is not from left to right.
- Same thing happens when we analyze sentences in a program.
- To the parse tree construction, the bottom-up procedure starts from isolated leaves, merges some of them to form a subtree, and eventually constructs the entire tree by placing a root.
- The parse tree generated by bottom-up has no difference with by top-down, the internal vertices are nonterminals and leaves are tokens.
- In fact, you have seen the similar procedure in Algorithm – the construction of Huffman code.

Bottom-Up Parsing

- Because the bottom-up parsing is an inverse of top-down paring, the basic strategy is to use the LHS of some production rules to replace the RHS after reading some input tokens.
- Thus, in the bottom-up parsing, there are two basic operations.
- *"**shift**"*
  - The parser needs to read more tokens from the input.
  - The tokens have already read are insufficient for any production rule.
- *"**reduce**"*
  - The parser has already read enough tokens.
  - It can replace the RHS by the LHS of some production rules.
- The parser is called ***shift-reduce*** parser.
- The parser also needs a stack to hold the tokens which are read but not consumed (reduced) yet.

Bottom-Up Parsing

- Let's look at a small example first. Given the following grammar and try to parse $abbcbcde$.

$$
\begin{aligned}
S &\rightarrow aABe \\
A &\rightarrow Abc \\
A &\rightarrow b \\
B &\rightarrow d
\end{aligned}
$$

No.	Stack	Input	Output	No.	Stack	Input	Output
0		$abbcbcde$		7	aAb	cde	shift
1	a	$bbcbcde$	shift	8	$aAbc$	de	shift
2	ab	$bcbcde$	shift	9	aA	de	reduce using $A \rightarrow Abc$
3	aA	$bcbcde$	reduce using $A \rightarrow b$	10	aAd	e	shift
4	aAb	$cbcde$	shift	11	aAB	e	reduce using $B \rightarrow d$
5	$aAbc$	$bcde$	shift	12	$aABe$		shift
6	aA	$bcde$	reduce using $A \rightarrow Abc$	13	S		reduce using $S \rightarrow aABe$

Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

No.	Stack	Input	Output
0		$abbcbcde$	

Parse tree:

No.	Stack	Input	Output
1	a	$bbcbcde$	shift

Parse tree:

a

No.	Stack	Input	Output
2	ab	$bcbcde$	shift

Parse tree:

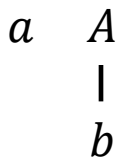
$a \quad b$

Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

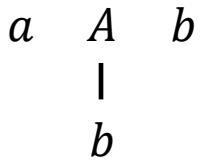
No.	Stack	Input	Output
3	aA	$bcbcde$	reduce using $A \rightarrow b$

Parse tree:



No.	Stack	Input	Output
4	aAb	$cbcde$	shift

Parse tree:

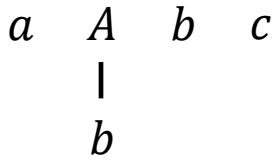


Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

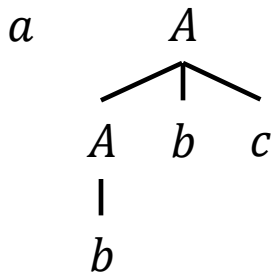
No.	Stack	Input	Output
5	$aAbc$	$bcde$	shift

Parse tree:



No.	Stack	Input	Output
6	aA	$bcde$	reduce using $A \rightarrow Abc$

Parse tree:

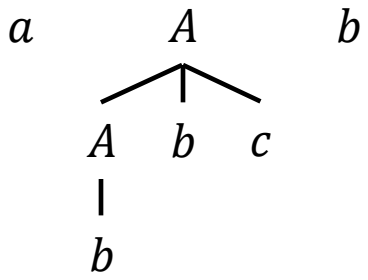


Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

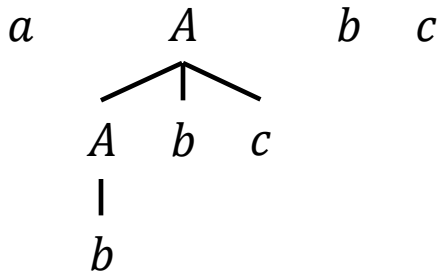
No.	Stack	Input	Output
7	aAb	cde	shift

Parse tree:



No.	Stack	Input	Output
8	$aAbc$	de	shift

Parse tree:

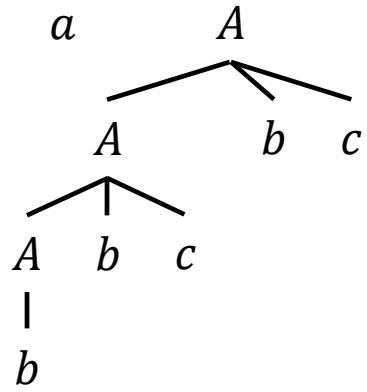


Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

No.	Stack	Input	Output
9	aA	de	reduce using $A \rightarrow Abc$

Parse tree:

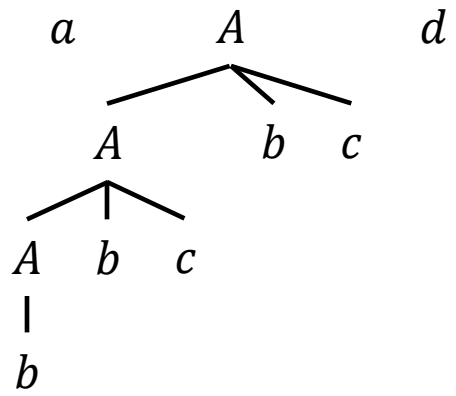


Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

No.	Stack	Input	Output
10	aAd	e	shift

Parse tree:

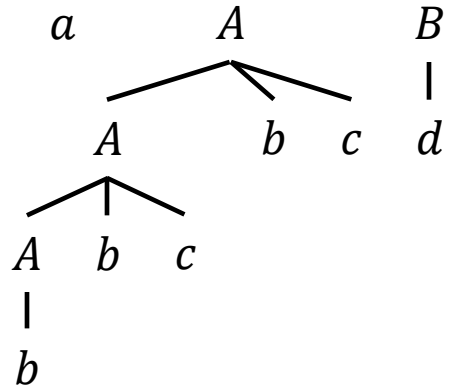


Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

No.	Stack	Input	Output
11	aAB	e	reduce using $B \rightarrow d$

Parse tree:

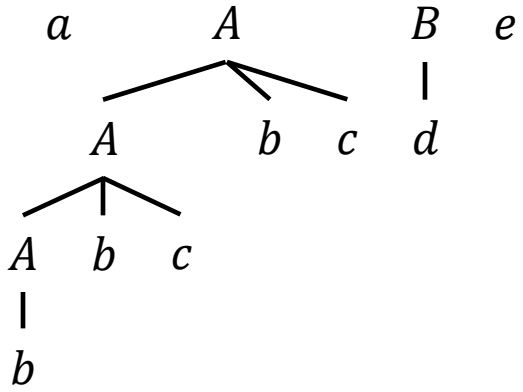


Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

No.	Stack	Input	Output
12	$aABe$		shift

Parse tree:

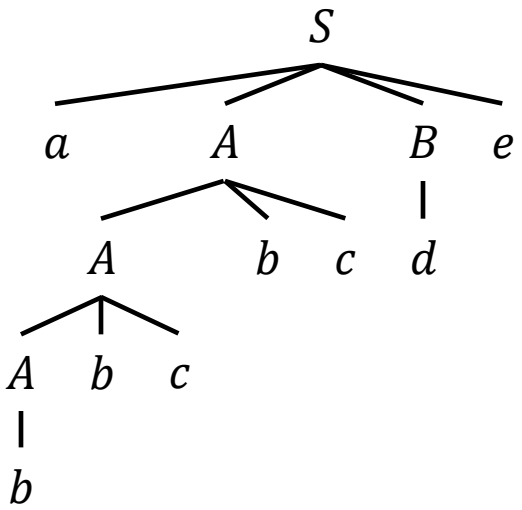


Bottom-Up Parsing

- Simultaneously, the parse tree is constructed.

No.	Stack	Input	Output
13	S		reduce using $S \rightarrow aABe$

Parse tree:



Bottom-Up Parsing

- The above example shows how bottom-up parsing works on a small instance.
- However, to formally define a bottom-up parser, we need to generalize this procedure to all grammars. Many details will be discussed.
- Even within this small example, careful reader may find the above procedure has some problems.
- In Step2, we read b. Then, we immediately reduce b to A in Step3.
- But the thing is different in Step4. We read b again, but the parser waits until it reads another c and reduces Abc to A.
- In general, similar to the predictive parser, there might be multiple options in bottom-up parsing. We need to clearly define which option is used under a certain condition.

LR Parser

- There are many different ways to solve the above problem, like using recursions (Same as what we did in the predictive parsing, recursions simply enumerate all possible options.)
- In this course, we only introduce $LR(k)$ parsers (k lookahead tokens, LR parser for short) because
  - LR parsers can be used to almost all (but not all) context-free grammars;
  - they are the most powerful non-backtracking shift-reduce parsers;
  - they can be implemented very efficiently; and
  - they are strictly more powerful than $LL(k)$ grammars.

  - When a reduction is performed, the RHS of the production rule is already on the stack, which is some additional information to help decision making.

LR Parsing Algorithm

- Similar to the predictive parser, the LR parsing algorithm is also define in the form of a push down automata.
- To describe the LR parsing algorithm, we first define ***configurations***, in the form:

$$\langle 0\ \underline{X_1 S_1}\ \underline{X_2 S_2}\ \cdots \underline{X_{m-1} S_{m-1}}\ \underline{X_m S_m}\ a_i a_{i+1} \cdots a_n \$ \rangle$$

  - $\langle\ \rangle$ indicates that this is a stream of symbols;
  - each X_i is a terminal or a nonterminal;
  - each 0 or S_i is a state in the PDA;
  - the underbar _ means that $X_i S_i$ form a pair, only for presenting purpose;
  - each a_i is an unprocessed input token; and
  - $ is an artificial token showing the end of input.
- Each configuration is the current content of the stack $0\underline{X_1 S_1}\ \cdots\ \underline{X_m S_m}$ glued together with the current content remained in the input $a_i a_{i+1} \cdots a_n$.
- S_m is the current state, and a_i is the next input token.

LR Parsing Algorithm

- The parser puts state symbols into the stack to speedup checking the content on the stack.
- Imaging that you want to check what's on the given stack. You need to pop everything out, see if the content is same as what you want, then push everything back. This is very inefficient.
- Thus, we use a state symbol to indicate the current content on the stack.
- This symbol is called state symbol because it is exactly same the states in a DFA.
- We can consider each state in a DFA means that "To reach this state, the input must be some specific strings."
- Also, each X_i is corresponds to an S_i, like a pair.

LR Parsing Table

- To parse a LR grammar, you are also given a parsing table.
- This parsing table again enumerates the actions the parser needs to take in all possible situations.
- Each row in the table represents a state in PDA.
- The columns are split into two individual fields: $ACTION$ and $GOTO$.
  - Each column in $ACTION$ represents a terminal.
  - Each column in $GOTO$ represents a nonterminal.

LR Parsing Table

- The value of the entry $ACTION[S_m, a_i]$ shows the action taken by the parser when the current state is S_m and the input token is a_i.
- The value are of two types:
  - $Shift\ S_i$, meaning that the parser pushes the next input token into the stack and move to the state S_i;
  - $Reduce\ r_i$, meaning that the parser reduces some (non)terminals using the production rule r_i.
- The value of the entry $GOTO[S_m, A]$ is a state symbol, S for example, meaning that the parser goes to the state S after it reduces the stack uses a production rule $A \rightarrow \beta$, for some β.

LR Parsing Algorithm

Algorithmically,

1. Put the special symbol $ at the end of the input.
2. Put state 0 at the bottom of the stack.
3. In each iteration, suppose the current configuration is $\langle 0\underline{X_1 S_1} \cdots \underline{X_m S_m} a_i \cdots a_n \$\rangle$, the current state is S_m, and the next input token is a_i
   1. If $ACTION[S_m, a_i]$ is "$shift\ S$", then the next configuration is $\langle 0\ \underline{X_1 S_1}\ \cdots\ \underline{X_m S_m}\ \underline{\boldsymbol{a_i S}} \cdots a_n \$\rangle$;
   2. If $ACTION[S_m, a_i]$ is "$reduce\ A \to \beta$", then the next configuration is $\langle 0\underline{X_1 S_1} \cdots \underline{X_{m-r} S_{m-r}}\ \underline{\boldsymbol{AS}} \cdots a_n \$\rangle$, where $r = |\beta|$ and $S = GOTO[S_{m-r}, A]$. At the same time, the parser outputs $A \to \beta$.
   3. If $ACTION[S_m, a_i]$ is "accept" and the current configuration is $\langle 0XS_1 \$\rangle$, where X is the start symbol of the grammar, the parser accepts the input.
   4. For all other cases, like $ACTION[S_m, a_i]$ is blank, the parser finds a syntax error and switch to error recovery.

Example

- Given the grammar
  1. $E \rightarrow E + T$
  2. $E \rightarrow T$
  3. $T \rightarrow T * F$
  4. $T \rightarrow F$
  5. $F \rightarrow (E)$
  6. $F \rightarrow id$
- Try to parse

  $id + id * id$

  (on the board)

State	*ACTION*						*GOTO*		
	id	$+$	$*$	$($	$)$	$\$$	E	T	F
0	$s5$			$s4$			1	2	3
1		$s6$				acc			
2		$r2$	$s7$		$r2$	$r2$			
3		$r4$	$r4$		$r4$	$r4$			
4	$s5$			$s4$			8	2	3
5		$r6$	$r6$		$r6$	$r6$			
6	$s5$			$s4$				9	3
7	$s5$			$s4$					10
8		$s6$			$s11$				
9		$r1$	$s7$		$r1$	$r1$			
10		$r3$	$r3$		$r3$	$r3$			
11		$r5$	$r5$		$r5$	$r5$			

Example

Stack	Input	Output
0	$id + id * id\$$	
$0\underline{id\ 5}$	$+id * id\$$	$shift\ 5$
$0\underline{F\ 3}$	$+id * id\$$	$reduce\ F \to id$
$0\underline{T\ 2}$	$+id * id\$$	$reduce\ T \to F$
$0\underline{E\ 1}$	$+id * id\$$	$reduce\ E \to T$
$0\underline{E\ 1}\ \underline{+\ 6}$	$id * id\$$	$shift\ 6$
$0\underline{E\ 1}\ \underline{+\ 6}\ \underline{id\ 5}$	$* id\$$	$shift\ 5$
$0\underline{E\ 1}\ \underline{+\ 6}\ \underline{F\ 3}$	$* id\$$	$reduce\ F \to id$
$0\underline{E\ 1}\ \underline{+\ 6}\ \underline{T\ 9}$	$* id\$$	$reduce\ T \to F$
$0\underline{E\ 1}\ \underline{+\ 6}\ \underline{T\ 9}\ \underline{*\ 7}$	$id\$$	$shift\ 7$
$0\underline{E\ 1}\ \underline{+\ 6}\ \underline{T\ 9}\ \underline{*\ 7}\ \underline{id\ 5}$	$\$$	$shift\ 5$
$0\underline{E\ 1}\ \underline{+\ 6}\ \underline{T\ 9}\ \underline{*\ 7}\ \underline{F\ 10}$	$\$$	$reduce\ F \to id$
$0\underline{E\ 1}\ \underline{+\ 6}\ \underline{T\ 9}$	$\$$	$reduce\ T \to T * F$
$0\underline{E\ 1}$	$\$$	$reduce\ E \to E + T$
$0\underline{E\ 1}$	$\$$	$accept$

End of lecture 7