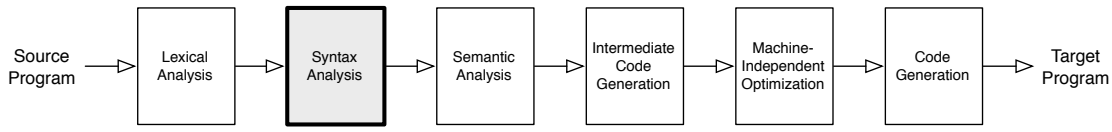
CS 301 Spring 2021 T4
Tutorial Assignment
22 April

Bottom-Up Parsing

1 Plan



This material focuses on *bottom-up parsing*, which constructs a parse tree starting from the leaves and working up toward the root. Bottom-up LR parsers can parse languages described by a larger class of grammars than top-down parsers, and they more easily handle grammar ambiguity of the form common in programming languages. (We will get experience with ambiguity when building our parser soon.) The algorithms for automatically building parsers from grammrs are detailed. Follow the algorithms in the book *carefully* for these exercises.

2 Reading

- Bottom-Up / LR Parsing:
  - Overview: Dragon 4.5,
  - LR parsing mechanics: Dragon 4.6 (see 4.4.2 for FIRST and FOLLOW), 4.7 – 4.7.3
  - Handing ambiguity: Dragon 4.8 – 4.8.2

  Alternative: EC 3.4 – 3.5 (see 103–107 *Backtrack-Free Parsing* within 3.3.1 for FIRST and FOLLOW)
- **Extra Depth:**
  - Error Recovery: Dragon 4.1.3 – 4.1.4, 4.8.3

Reminder: Dragon tends toward mathematical notations. EC tends toward imperative pseudocode. Use the one that's easiest for you to parse (pun intended).

3 Exercises

1. (Adapted from Dragon Exercise 4.5.3)

   (a) Given the following grammar, in which `0` and `1` are terminals:

   $$\begin{array}{rcl} S & \rightarrow & \texttt{0}\ S\ \texttt{1} \\ & | & \texttt{0}\ \texttt{1} \end{array}$$

   Show the series of larger and larger parse trees that would be built in a bottom-up parse of the string `000111` with this grammar.

   (b) Given the following grammar, in which `a`, `*`, and `+` are terminals:

   $$\begin{array}{rcl} S & \rightarrow & S\ S\ \texttt{+} \\ & | & S\ S\ \texttt{*} \\ & | & \texttt{a} \end{array}$$

   Show the series of larger and larger parse trees that would be built in a bottom-up parse of the string `aaa*a++` with this grammar.

2. The following grammar describes the language of regular expressions. Terminals are quoted to avoid confusing them with the grammar metasyntax. The terminals include '`|`' (small bar), '`(`' and '`)`' (parentheses), '`*`' (star), and 'ϵ' (epsilon), as well as the characters '`a`' through '`e`'.

$$R \rightarrow R \text{ `|' } R \;\Big|\; R\,R \;\Big|\; R \text{ `*'} \;\Big|\; \text{`(' } R \text{ `)'} \;\Big|\; \text{`}\epsilon\text{'} \;\Big|\; \text{`a'} \;\Big|\; \text{`b'} \;\Big|\; \text{`c'} \;\Big|\; \text{`d'} \;\Big|\; \text{`e'}$$

This grammar is ambiguous. The ambiguity can be resolved by these precedence rules: Kleene star ('`*`') has higher precedence than concatenation; concatenation has higher precedence than alternation ('`|`').

(a) Write an LR grammar that accepts the same language, respects the desired operator precedence, and ensures that alternation is left-associative, but concatenation is right-associative. (Note: You need not prove that your grammar is LR.)

(b) Write the parse tree for the input "`a|bc*d|e`" using the LR grammar.

3. Compute the FIRST and FOLLOW sets for this grammar:

$$\begin{array}{lcl} S & \rightarrow & B\ C\ \mathbf{z} \\ B & \rightarrow & \mathbf{x}\ B \mid D \\ C & \rightarrow & \mathbf{u}\ E \\ D & \rightarrow & \mathbf{y}\ D \mid \epsilon \\ E & \rightarrow & \mathbf{v} \mid \epsilon \end{array}$$

4. Compute the FIRST and FOLLOW sets for the following grammar of statements:

$$\begin{array}{rcl} \mathit{Stmt} & \rightarrow & \texttt{if E then } \mathit{Stmt}\ \mathit{StmtTail} \\ & \mid & \texttt{while E } \mathit{Stmt} \\ & \mid & \{\ \mathit{List}\ \} \\ & \mid & \texttt{S} \\ \\ \mathit{StmtTail} & \rightarrow & \texttt{else } \mathit{Stmt} \\ & \mid & \epsilon \\ \\ \mathit{List} & \rightarrow & \mathit{Stmt}\ \mathit{ListTail} \\ \\ \mathit{ListTail} & \rightarrow & ;\ \mathit{List} \\ & \mid & \epsilon \end{array}$$

Unlike Java (and like ML), semicolons in the syntax defined by this grammar *separate* rather than *terminate* statements. You can assume `E` and `S` are terminals that represent other expression and statement forms that we do not currently care about.

5. (Adapted from Dragon Exercises 4.6.2–4.6.3) Consider the following familiar grammar, in which `a`, `*`, and `+` are terminals:

$$\begin{array}{lcl} S & \rightarrow & S\ S\ \texttt{+} \\ & \mid & S\ S\ \texttt{*} \\ & \mid & \texttt{a} \end{array}$$

Note that Figure 1 may be a helpful format reference for this exercise: it shows an LR(0) automaton and SLR parsing table for a later exercise.

(a) Construct the FIRST and FOLLOW sets for this grammar.

(b) Construct the LR(0) sets of items and the LR(0) automaton for the grammar, as augmented with $S' \to S$. (See Dragon 4.6.2.)

(c) Construct the SLR parsing table for this LR(0) automaton/grammar, including, for each state:

- the ACTION for each terminal; and
- the GOTO for each non-terminal.

(See Dragon 4.6.4.)

(d) Demonstrate the behavior of this parsing table on the input `aa*a+`. Show the steps of the parser in the following format.

- The Stack column indicates the stack of statues.
- The Symbols column indicates the workspace where symbols are shifted and reduced.
- The Input column shows the remaining input (with the end marked by $).
- The Action column indicates what step will be taken to get from the status shown in this row to produce the status in the next row.

Stack	Symbols	Input	Action
0		`aa*a+` $	shift
...	...	...	...

6. Consider the following grammar:

$$E \to id \mid id\ (\ E\ ) \mid E + id$$

(a) Build the LR(0) automaton for this grammar.

(b) Show that the grammar is not an LR(0) grammar by building the parsing table. (LR(0) parsing table construction is left implicit in the text — however, it is essentially Algorithm 4.46, where Rule 2(b) is applied for all a, rather than for all a in FOLLOW (A).)

(c) Is this an SLR grammar? Give evidence.

(d) Is this an LR(1) grammar? Give evidence.

7. Consider the grammar of matched parentheses:

$$\begin{aligned} A &\to (\ A\ )\ A \\ &\mid \epsilon \end{aligned}$$

(a) Construct the LR(1) automaton.

(b) Build the LR(1) parsing table to show that the grammar is LR(1).

(c) Is the grammar LR(0)? Justify your answer.

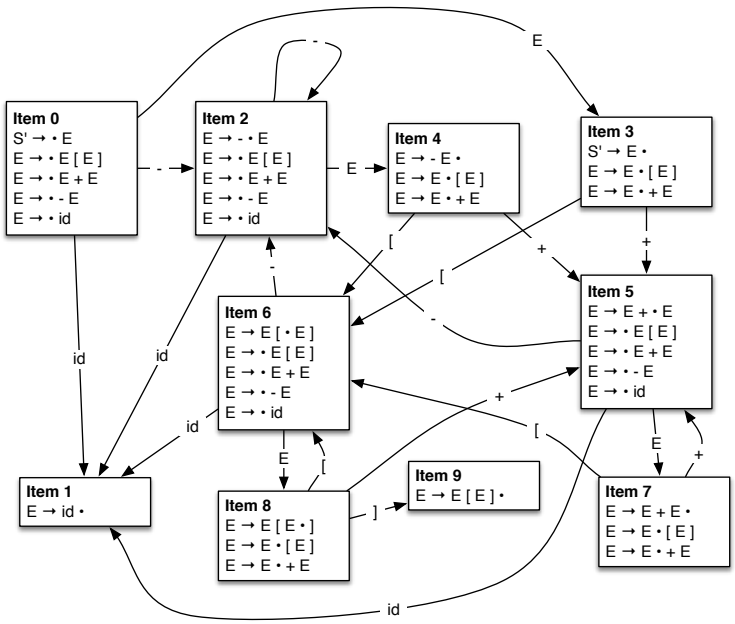
8. The following grammar describing expressions over addition, negation, and array accesses is ambiguous. (Parenthesized numbers to the right label the productions; they are not part of the grammar.)

$$\begin{aligned} E &\to E[E] && (1) \\ &\mid E + E && (2) \\ &\mid -E && (3) \\ &\mid \texttt{id} && (4) \end{aligned}$$

To generate an LR parser for this grammar, we could rewrite the grammar. It is also possible to eliminate the ambiguity directly in the parsing table by exploiting precedence and associativity rules.

Figure 1 shows the LR(0) automaton and SLR parsing table for this grammar.

Figure 1: LR(0) automaton, FIRST and FOLLOW sets, and parsing table for exercise 8



X	FIRST(X)	FOLLOW(X)
S'	\$	\$
E	`id`, $-$	\$, [,], $+$

	Action						*Goto*	
State	[	]	$+$	`id`	$-$	\$	S'	E
0				$s1$	$s2$			3
1	$r4$	$r4$	$r4$			$r4$		
2				$s1$	$s2$			4
3	$s6$		$s5$			acc		
4	$s6/r3$	$r3$	$s5/r3$			$r3$		
5				$s1$	$s2$			7
6				$s1$	$s2$			8
7	$s6/r2$	$r2$	$s5/r2$			$r2$		
8	$s6$	$s9$	$s5$					
9	$r1$	$r1$	$r1$			$r1$		

(a) Given that + is left-associative and has a lower precedence than unary negation, and that negation has lower precedence than array accesses, eliminate the conflicts in the SLR table by removing actions from the problematic table entries. Justify how you resolved conflicts.

(b) Show how your resulting parser handles the input `id` + `id`[`id`] + `id`.

9. **Extra Depth:** Compare the LL(1) and LR(1) parsing techniques on the basis of expressiveness, error reporting, and understandability (for the programming language implementer), indicating their advantages and disadvantages.

10. **Extra Depth:** Here is a grammar similar to the one used to consider error recovery in LL parsers:

$$
\begin{array}{rcll}
\mathit{Stmt} & \rightarrow & \texttt{if E then } \mathit{Stmt} & (1) \\
 & | & \texttt{if E then } \mathit{Stmt} \texttt{ else } \mathit{Stmt} & (2) \\
 & | & \texttt{while E } \mathit{Stmt} & (3) \\
 & | & \{\ \mathit{List}\ \} & (4) \\
 & | & \texttt{S} & (5) \\
 & & & \\
\mathit{List} & \rightarrow & \mathit{List}\ ;\ \mathit{Stmt} & (6) \\
 & | & \mathit{Stmt} & (7)
\end{array}
$$

Figure 2 shows the LR(0) automaton and parsing table for this grammar, with the dangling-else ambiguity resolved in the usual way. I have introduced the extra production $S' \rightarrow \mathit{Stmt}$.

(a) Implement error correction by filling in the blank entries in the parsing table with extra reduce actions or suitable error-recovery routines.

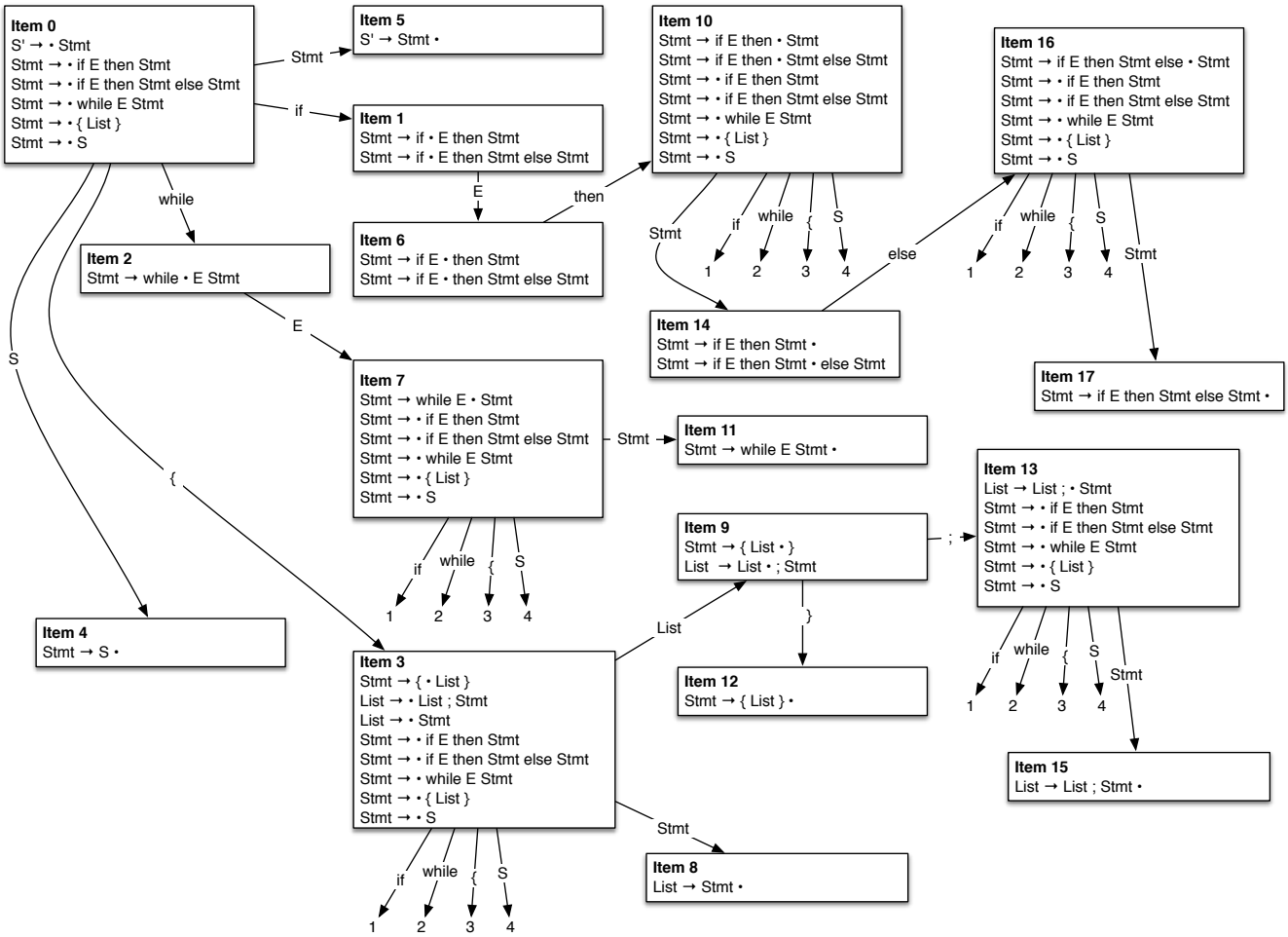
(b) Describe the behavior of your parser on the following two inputs:

- `if E then S ; if E then S }`
- `while E { S ; if E S ; }`

11. **Extra Depth:** Bottom-up LR parsers are still widely used, but there has been a resurgence of interest in other top-down techniques such as parser combinators, parsing expression grammars, and variants of LL parsers (*e.g.*, $LL(*)$, $ALL(*)$). Some more recent top-down techniques avoid key limitations of top-down parsing for most reasonable programming languages in practice. If you are curious, check out some of these papers:

- *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.*
  Bryan Ford. POPL 2004.
  `https://doi.org/10.1145/964001.964011`
- *LL(*): The Foundation of the ANTLR Parser Generator.*
  Terrence Parr, Kathleen Fisher. PLDI 2011.
  `https://doi.org/10.1145/1993498.1993548`
- *Adaptive LL(*) Parsing: the Power of Dynamic Analysis.*
  Terrence Parr, Sam Harwell, Kathleen Fisher. OOPSLA 2014.
  `https://doi.org/10.1145/2660193.2660202`
- ANTLR: `https://www.antlr.org/`

Figure 2: LR(0) automaton and parsing table for exercise 10



	Action										*Goto*	
State	`if`	`E`	`then`	`else`	`while`	`S`	{	}	;	$	*Stmt*	*List*
0	$s1$				$s2$	$s4$	$s3$				5	
1		$s6$										
2		$s7$										
3	$s1$				$s2$	$s4$	$s3$				8	9
4	$r5$	$r5$	$r5$	$r5$	$r5$	$r5$	$r5$	$r5$	$r5$	$r5$		
5										acc		
6			$s10$									
7	$s1$				$s2$	$s4$	$s3$				11	
8	$r7$	$r7$	$r7$	$r7$	$r7$	$r7$	$r7$	$r7$	$r7$	$r7$		
9								$s12$	$s13$			
10	$s1$				$s2$	$s4$	$s3$				14	
11	$r3$	$r3$	$r3$	$r3$	$r3$	$r3$	$r3$	$r3$	$r3$	$r3$		
12	$r4$	$r4$	$r4$	$r4$	$r4$	$r4$	$r4$	$r4$	$r4$	$r4$		
13	$s1$				$s2$	$s4$	$s3$				15	
14	$r1$	$r1$	$r1$	$s16$	$r1$	$r1$	$r1$	$r1$	$r1$	$r1$		
15	$r6$	$r6$	$r6$	$r6$	$r6$	$r6$	$r6$	$r6$	$r6$	$r6$		
16	$s1$				$s2$	$s4$	$s3$				17	
17	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$		