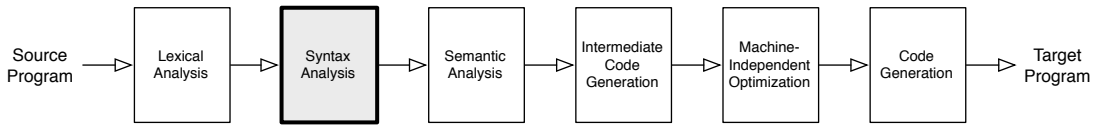
HW 3: Bottom-Up Parsing Techniques

CSCI 434T
Spring, 2019

Overview



This week focuses on a second form of parsing, called bottom-up parsing. In contrast to top-down parsing, this approach constructs the parse tree from the leaves up towards the root. As you will see, this is quite a powerful technique. For example, bottom-up LR parsers can parse languages described by a much larger class of grammars than LL parsers, and they more easily handle grammar ambiguity of the form common in programming languages. (More on this second point next week when we build the parser for IC...) As I'm sure you observed last week with LL parsing, the algorithms for building parsers can be quite detailed. I strongly encourage you to follow the algorithms in the book *carefully* as you work through these problems. I have also provided some optional reading and problems that look more deeply at syntax error recovery.

Readings

- Dragon 4.5 – 4.6, 4.7 – 4.7.3, 4.8.1 – 4.8.3
- (primarily the early sections) Michael Burke and Gerald Fisher, *A practical method for LR and LL syntactic error diagnosis and recovery*, 1987. On the web site.

Exercises

1. Dragon 4.5.3
2. The following grammar describes the language of regular expressions:

$$R \quad \rightarrow \quad R\ bar\ R \ \mid \ R\ R \ \mid \ R\ star \ \mid \ (\ R\ ) \ \mid \ \epsilon \ \mid \ letter$$

   where *bar*, *star*, *letter*, '(', and ')' are all terminals. This is an ambiguous grammar. The Kleene star operation has higher precedence than concatenation; and, in turn, concatenation has higher precedence than alternation.

   (a) Write a LR grammar that accepts the same language, respects the desired operator precedence, and is such 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 expression $a|bc*d|e$ using the LR grammar.

3. Dragon 4.6.2. You will find it useful to construct the LR(0) automaton while you are building the SLR items and the parsing table.
4. Dragon 4.6.3
5. Consider the following grammar:

$$E \quad \rightarrow \quad id \ \mid \ id\ (\ E\ ) \ \mid \ E\ +\ id$$

(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.

6. Consider the grammar of matched parentheses:

$$A \rightarrow (A) A \mid \epsilon$$

(a) Construct the LR(1) automaton.

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

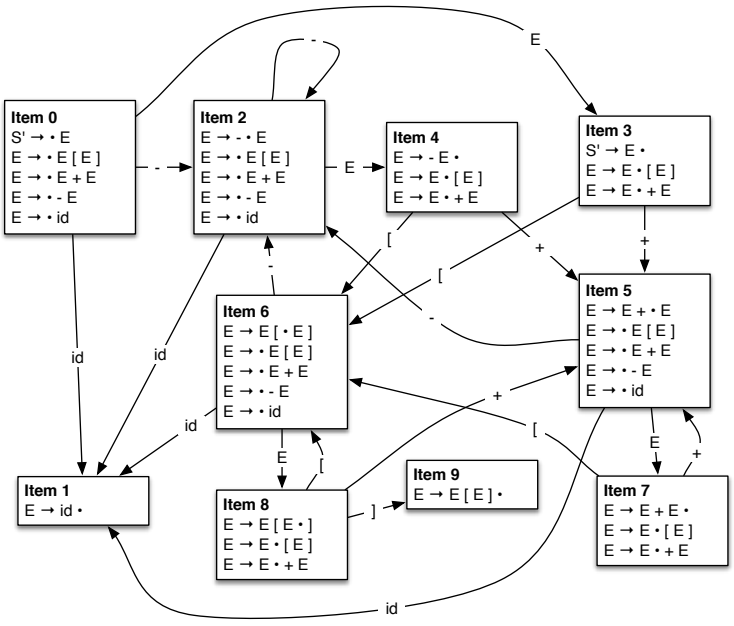
(c) Is the grammar LR(0)? Justify your answer.

7. The following grammar describing expressions over addition, negation, and array accesses is ambiguous:

$$\begin{array}{rcll} E & \rightarrow & E[E] & (1) \\ & | & E + E & (2) \\ & | & -E & (3) \\ & | & \texttt{id} & (4) \end{array}$$

To generate an LR parser for this grammar, we could rewrite the grammar. However, it is also possible to directly eliminate the ambiguity in the parsing table, taking advantage of precedence and associativity rules.

Here are the LR(0) automaton and SLR parsing table for this grammer:



X	$First(X)$	$Follow(X)$
S'	\$	\$
E	id, $-$	\$, [,], +

Parsing Table:

	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.

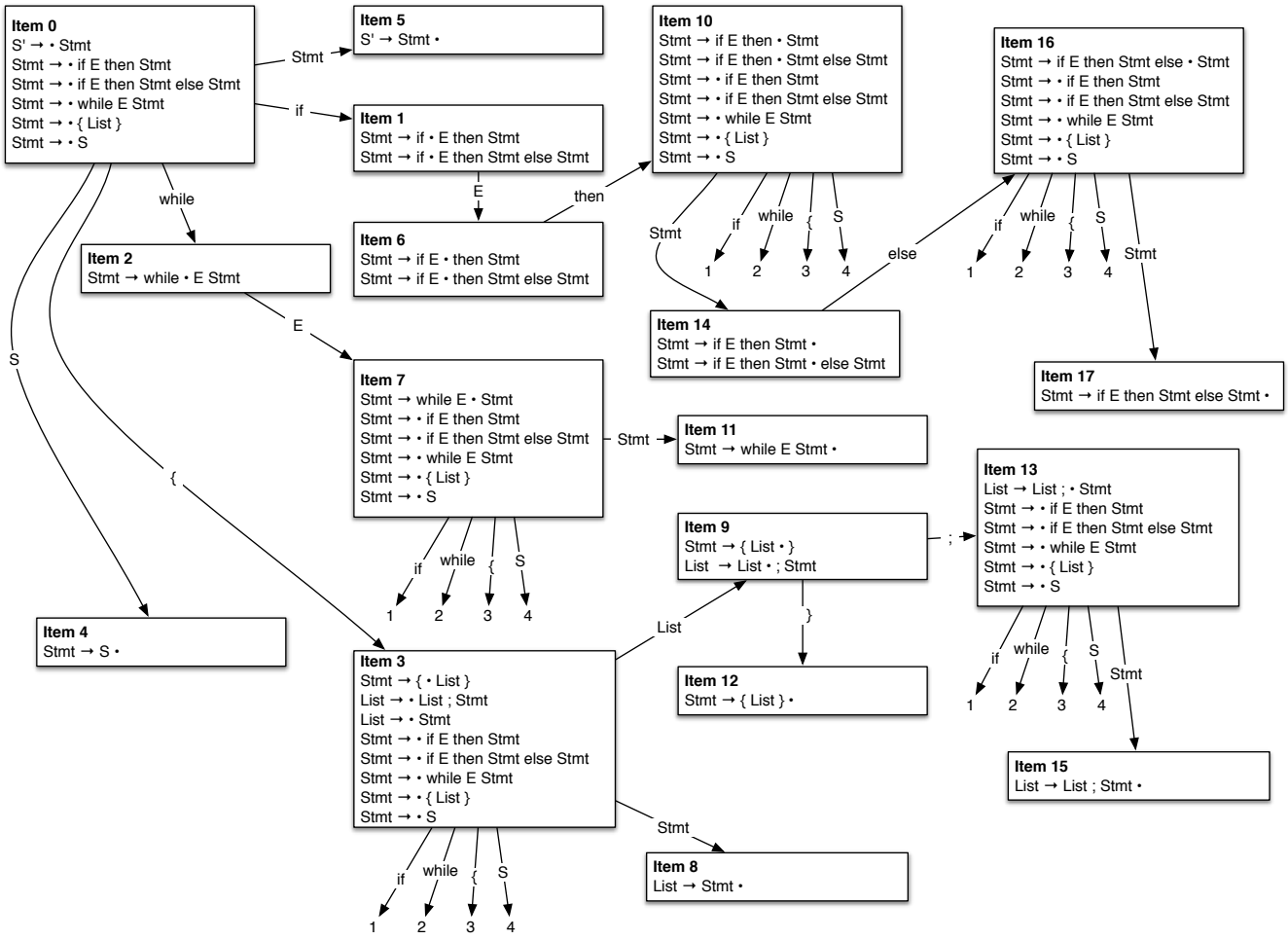
(b) Show how your resulting parser handles the input `id` + `id`[`id`] + `id`.

8. Here is a grammar similar to the one used to look at error recovery in LL parsers:

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

Here is 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

$$S' \rightarrow \mathit{Stmt}$$



Parsing Table:

	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	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$	$r2$		

(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 ; }`

9. **(Read enough of Burke-Fisher to get the basic idea of what they propose.)** The Burke-Fisher paper describes a different approach to syntax error recovery for parsers.

(a) What is their basic approach, and how does it differ from what you did last week (or in the previous problem)? In particular, how does their error handling fit into the general parsing algorithm? You may wish to focus only on the early sections involving single-token recovery.

(b) Are there advantages or disadvantages to this approach? Which would you prefer to use while developing a compiler for a large language?