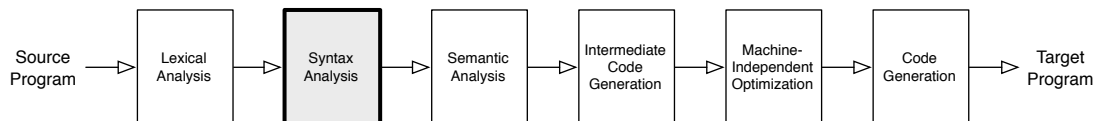


## Plan

---



This week we focus 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 much larger class of grammars than top-down LL 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.) We also consider how to improve error-reporting during parsing.

As with LL parsing, the algorithms for building parsers are quite detailed. Follow the algorithms in the book *carefully* as you work through these exercises. I will spend some time on Friday, February 12, reviewing top-down LL parsing and giving a sneak peek of bottom-up LR parsing, since these topics are tricky.

Optional: Bottom-up LR parsers are still widely used, but there has been some resurgence of interest in other top-down techniques such as parser combinators and variants of LL parsers (*e.g.*, ALL(\*)). Some relatively recent top-down techniques avoid key limitations of top-down parsing for most reasonable programming languages in practice. Ask if you are curious.

## Readings

---

- Dragon 4.5 – 4.6, 4.7 – 4.7.3, 4.8.1 – 4.8.3, 4.1.3 – 4.1.4 (for perspective on error recovery)  
Alternative: EC 3.4 – 3.5

Remember, copies of Dragon and IC are available in the bookshelf outside my door. Dragon’s algorithms tend toward math-style notations. EC’s algorithms tend toward imperative pseudocode. If you find Dragon’s algorithms and descriptions hard to parse (pun intended), take a look at EC in case its formulation makes more sense.

## Exercises

---

1. Review the computation of FIRST and FOLLOW from last week’s exercise 7b and relevant parts of 8a (before the LL-specific table building). Try to construct these on your own if you did not last week. Use the solutions I provided to check your understanding.
2. Dragon 4.5.3
3. The following grammar describes the language of regular expressions:

$$R \rightarrow R \text{ bar } R \mid R R \mid R \text{ star } \mid ( R ) \mid \epsilon \mid \text{char}$$

where *bar*, *star*, *char*, “(”, 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 an 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.

4. 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.
5. Dragon 4.6.3
6. Consider the following grammar:

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

- (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:

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

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

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

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:

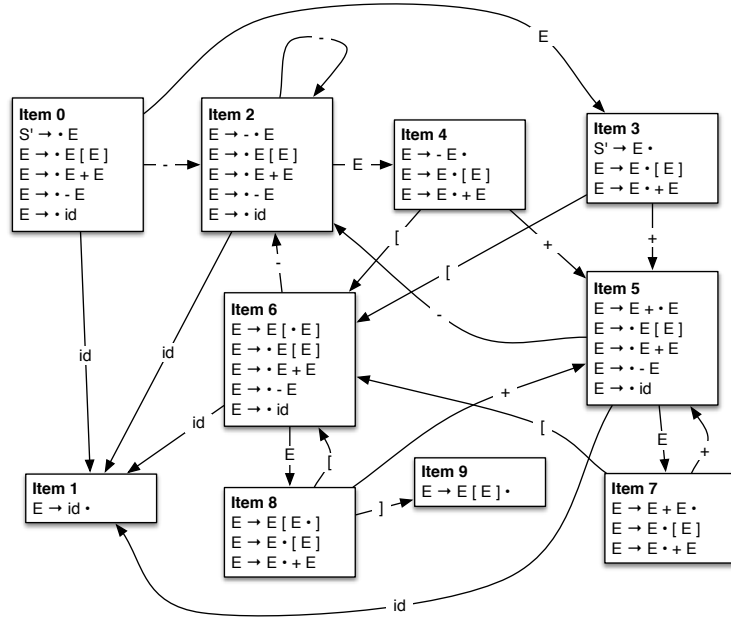
- (a) Given that  $+$  is left-associative and has a lower precedence than unary negation, and that negation has lower precedence than array accesses, eliminate the conflicts in the SLR table by removing actions from the problematic table entries. Justify how you resolved conflicts.
  - (b) Show how your resulting parser handles the input `id + id[id] + id`.
9. Here is a grammar similar to the one used to look at error recovery in LL parsers:

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

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  $S' \rightarrow Stmt$

- (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 ; }`

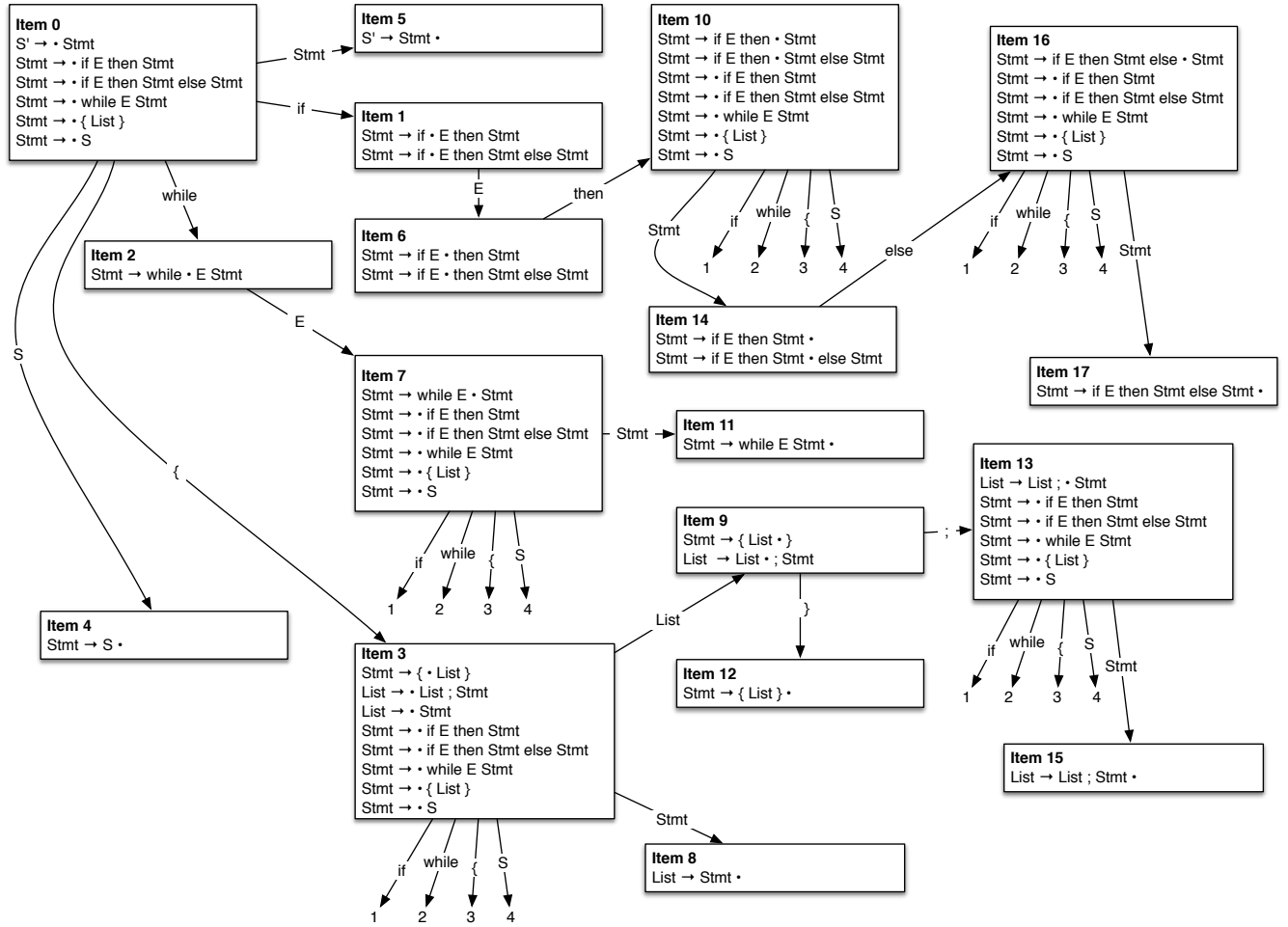
Figure 1: LR(0) automaton, FIRST and FOLLOW sets, and parsing table for exercise 8



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

State	Action						Goto	
	[	]	+	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		

Figure 2: LR(0) automaton and parsing table for exercise 9



State	Action										Goto	
	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		