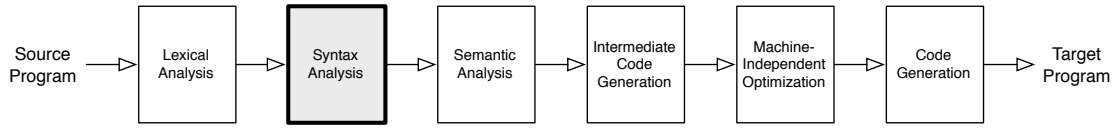
CS 301 Spring 2019
Tutorial Assignment
19 February (Reschedule)

Bottom-Up Parsing

1 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, these algorithms for building parsers are detailed. Follow the algorithms in the book *carefully* for these exercises.

2 Reading

- 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

Reminder: Dragon tends toward mathematical notations. EC tends toward imperative pseudocode. Use the one that's easiest for you to parse (pun intended). Copies of both are available in the lab windowsill shelf.

3 Exercises

1. Review the computation of FIRST and FOLLOW from last week's exercise 7b and 8a. (Skip LL table building.)
2. Dragon 4.5.3
3. The following grammar describes the language of regular expressions:

   $$R \;\rightarrow\; R\ bar\ R \;\mid\; R\ R \;\mid\; R\ star \;\mid\; (\ R\ ) \;\mid\; \epsilon \;\mid\; char$$

   where *bar*, *star*, *char*, "(", and ")" are all terminals. This grammar is ambiguous. 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 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 a simple grammar for pointer expressions in C, consisting of pointer dereference expressions, address-of expressions, assignments, and field accesses:

   $$E \;\rightarrow\; *E \;\mid\; \&E \;\mid\; E = E \;\mid\; E \texttt{->} E \;\mid\; id$$

This is an ambiguous grammar. We would like to write an unambiguous grammar for the same language, such that field accesses E`->`E have higher precedence than dereferences and address-of expressions, and all of these have higher precedence than assignments.

(a) Write an LL(1) grammar which accepts the same language and has the desired operator precedence. Show the LL(1) parsing table for this grammar.

(b) Write an LR(1) grammar which accepts the same language, respects the desired operator precedence, and is such that assignments are right-associative, and field accesses are left-associative.

(c) Write the parse tree for the expression `**a->b->c = &*d` using the LR(1) grammar.

(d) One problem with the grammar above is that it models a superset of the valid C expressions. For instance, `&a = b->*c` is an invalid expression. We therefore impose the following conditions:

- only a location (a dereference or an identifier) can occur on the right-hand side of an assignment;
- only a location can occur in the address-of construct;
- the address-of expression can occur only in dereferences or the right side of an assignment; and
- the expression in the right-hand side of a field access must be an identifier.

Write a LR(1) grammar which precisely accepts this language and has the desired precedence and associativity of operators.

7. 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.

8. 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.

9. 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.)

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

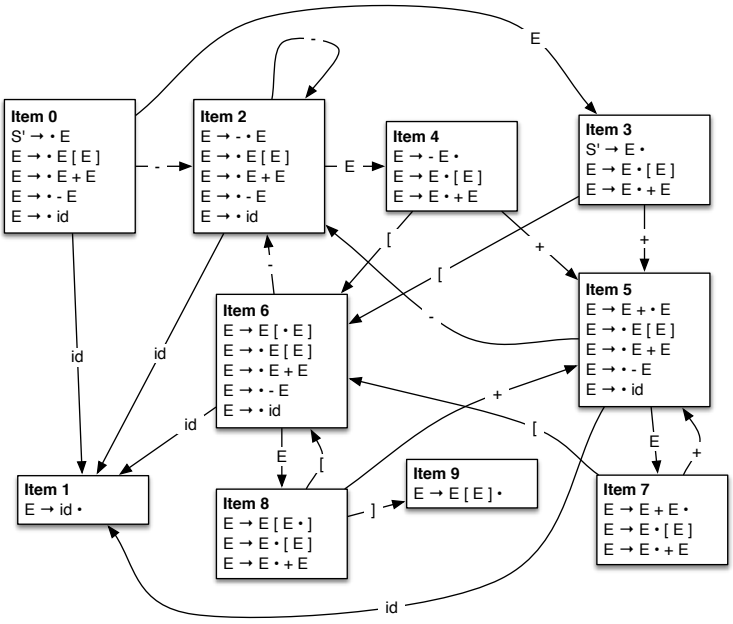
To generate an LR parser for this grammar, we could rewrite the grammar. It is also possible to eliminate the ambiguity directly in the parsing table by exploiting precedence and associativity rules.

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 $\texttt{id} + \texttt{id}[\texttt{id}] + \texttt{id}$.

Figure 1: LR(0) automaton, FIRST and FOLLOW sets, and parsing table for exercise 9



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	$r1$	$r1$	$r1$			$r1$		

10. 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.

11. Here is a grammar similar to the one used to consider error recovery in LL parsers:

Stmt	→	`if E then` *Stmt*	(1)
	|	`if E then` *Stmt* `else` *Stmt*	(2)
	|	`while E` *Stmt*	(3)
	|	{ *List* }	(4)
	|	`S`	(5)
List	→	*List* ; *Stmt*	(6)
	|	*Stmt*	(7)

    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 $S' \to Stmt$.

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

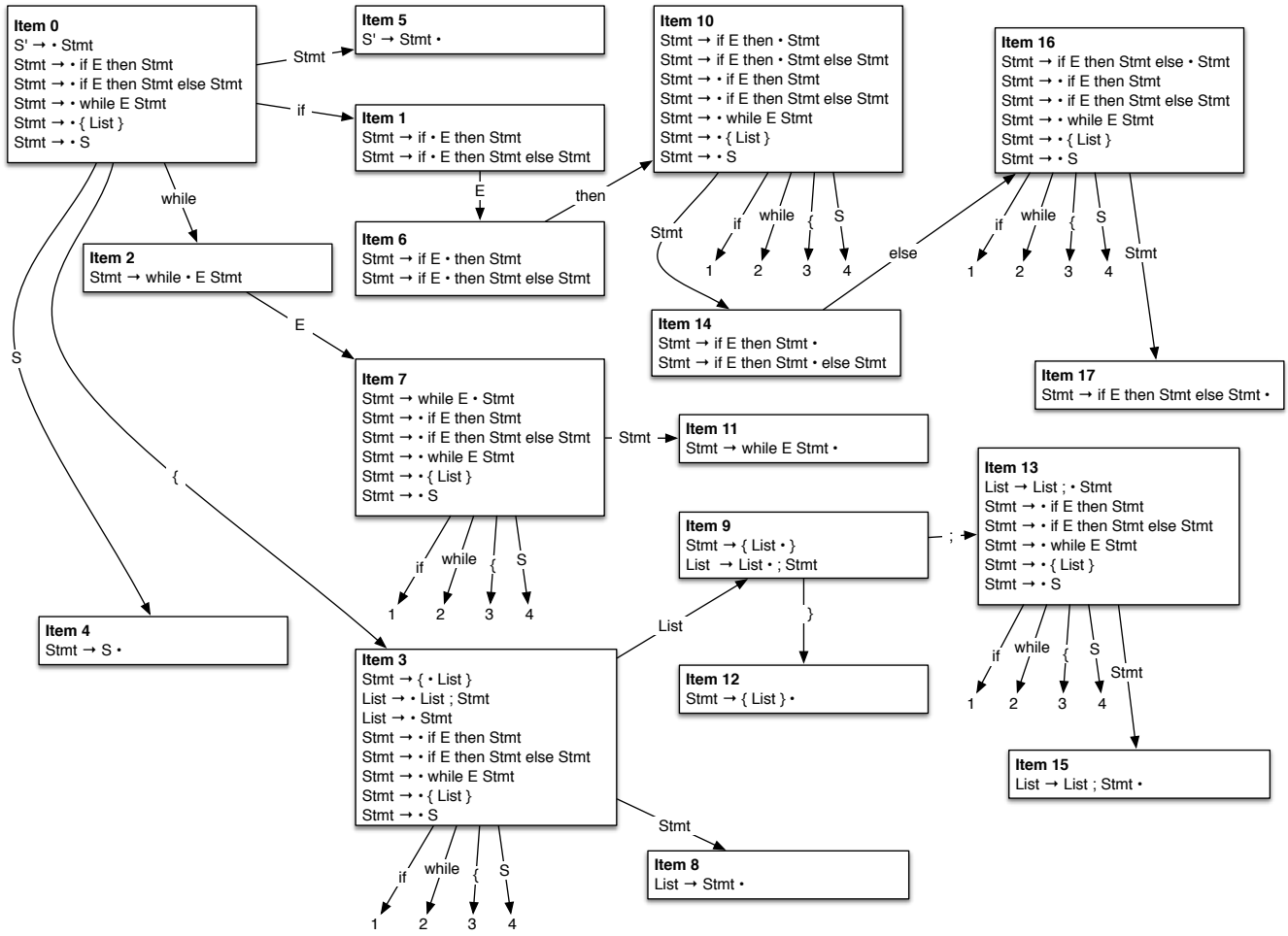
    (b) Describe the behavior of your parser on the following two inputs:

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

12. **Optional:** 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 11



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