The second example of constructing a bottom-up parser

- Given the following production rules that define expressions:

  5. E' → <expr>
  6. <expr> → <expr> + <term>
  7. → <expr> - <term>
  8. <term> → (<expr>)
  9. → int
  10. <expr> → <term>

  - NOTE: we construct the LR(0) states next, but leave the parsing table construction and parsing examples to students as exercises.

We derive the LR(0) states

- S1: E' → . <expr> (goto S2, S3, S4, S5)
  <expr> → . <expr> + <term>
  <expr> → . <expr> - <term>
  <expr> → . <term>
  <term> → . int
  <term> → . (<expr>)
- S2:E' → <expr> . (accept?) (goto S6, S7)
  <expr> → <expr> . + <term>
  <expr> → <expr> . - <term>
- S3: <expr> → <term> . (r10)
- S4: <term> → int . (r9)
- S5: <term> → (. <expr>) (goto S8, S4, S5, S3)
  <expr> → . <expr> + <term>
  <expr> → . <expr> - <term>
  <expr> → . <term>
  <term> → . int
  <term> → . (<expr>)
- S6: <expr> → <expr> + . <term> (goto S9, S4, S5)
  <term> → . int
  <term> → . (<expr>)
- S7: <expr> → <expr> - . <term> (goto S10, S4, S5)
  <term> → . int
  <term> → . (<expr>)
- S8: <term> → (<expr> .) (goto S6, S7, S11)
  <expr> → <expr> . + <term>
  <expr> → <expr> . - <term>
- S9: <expr> → <expr> + <term> . (r6)
- S10: <expr> → <expr> - <term> . (r7)
- S11: <term> →(<expr>) . (r8)

The parsing table we constructed in class

States	Action					Goto		
	id	=	int	;	$	L	<stmt>	<expr>
1	s2					g8	g7	
2		s3						
3			s4					g5
4	r5	r5	r5	r5	r5			
5				s6				
6	r4	r4	r4	r4	r4			
7	s2				r2	g9	g7	
8					acc			
9	r3	r3	r3	r3	r3			

NOTE: r5 means using rule 5 to reduce; acc means accept the input;
From this table, one can draw the state diagram that has circles and edges, which we omit here.

Details in each state

- We constructed our parsing table based on “items” we found in each state and deduce all possible transitions.
- If a transition leads to a set of items (including the closure) that hasn’t appeared before, we create a new state.
- All transitions from the same state and labeled by the same symbol must have the same state as the target!
- State numbering is arbitrary. In the future, we examine each state, for which we haven’t examined the transitions yet, one by one, creating new states as needed. We keep a work list to store the new states. Removing a state from the work list at a time.
- For this example, we introduce new states in “depth-first” manner instead, in order to explain the state transitions in terms of an input example.
- The state tables built “depth-first” and “breadth-first” will be equivalent, differing only in the state numbering.

Applying the parsing table to an input example

- The parser maintains a parsing stack to store the history of shifts , reduces and gotos.
- Given input: “a = 3; b = 0;”
  - Stack -- next token position
  - 1 -- ^ a = 1; b = 0;
  - 1 2 -- a ^ = 1; b = 0;
  - 1 2 3 -- a = ^ 1; b = 0;
  - 1 2 3 4 (r5) -- a = 1 ^ ; b = 0;
  - 1 2 3 (g5)
  - 1 2 3 5
  - 1 2 3 5 6 (r4) -- a = 1 ; ^ b = 0;
  - 1 (g7)
  - 1 7
  - 1 7 2 -- a = 1 ; b ^ = 0;
  - 1 7 2 3 -- a = 1; b = ^ 0;
  - 1 7 2 3 4 (r5) -- a = 1; b = 0 ^ ;
  - 1 7 2 3 (g5)
  - 1 7 2 3 5
  - 1 7 2 3 5 6 (r4) -- a = 1; b = 0 ; ^
  - 1 7 (g7)
  - 1 7 7 (r2)
  - 1 7 (g9)
  - 1 7 9 (r3)
  - 1 (g8)
  - 1 8 (acc)