Bottom-up parsing

How to construct a parser for any SLR(1) grammar.

More powerful than LL(1) grammar.

Bottom-up parsing

Bottom-up (LR) parsers:

- Produce rightmost derivation.
- Work from bottom (leaves) of parse tree upwards.
- Decide late which production to use by keeping track of all. Decide only after whole right side of production has been found.
- More powerful than top-down (LL) parsers: there are grammars that are LR(*k*) but not LL(*k*) (for any *k*).

LR grammars

- LR(0): simplest but not powerful.
- SLR(1): more powerful than LR(0) but still simple.
- LALR(1): more powerful and complex than SLR(1). Can handle most artificial languages.
- LR(1): more powerful than LALR(1) but requires large tables.

We will look at SLR(1) in detail: a minor extension to LR(0).

LR(0) automata

- An LR(0) or SLR(1) grammar can be converted to a finite automaton.
- LR(0) automaton keeps track of current position in all productions as it reads terminal symbols from the input.
- Cf. DFA for lexical analysis: keeps track of all possible tokens as it reads characters. But now we also need a stack.
- **Example**: previous grammar in simple (left-recursive) form. We always add a new start symbol, S' and a production $S' \rightarrow E\ \$$

0: $S' \rightarrow E\$$
1: $E \rightarrow M$
2: $E \rightarrow E + M$
3: $M \rightarrow F$
4: $M \rightarrow M * F$
5: $F \rightarrow x$
6: $F \rightarrow (E)$

LR(0) automaton construction

- **State**. Every state contains a set of “items”. Item is a production with a position in its right side (indicated by “.”).
- **Initial state**. Initial state contains production for new start symbol with position at beginning of its right side.

$$\boxed{S' \to .\, E\ \$}$$

- **Closure**. Every state automatically contains closure of the set of items in it:

$$\boxed{\begin{array}{l} S' \to .\, E\ \$ \\ E \to .\, M \\ E \to .\, E + M \\ M \to .\, F \\ M \to .\, M * F \\ F \to .\, x \\ F \to .\, (\, E\,) \end{array}}$$

Closure computation

Repeat (until J does not change):

For each item $A \rightarrow \alpha \,.\, B\,\gamma$ in J:

For each production $B \rightarrow \beta$:

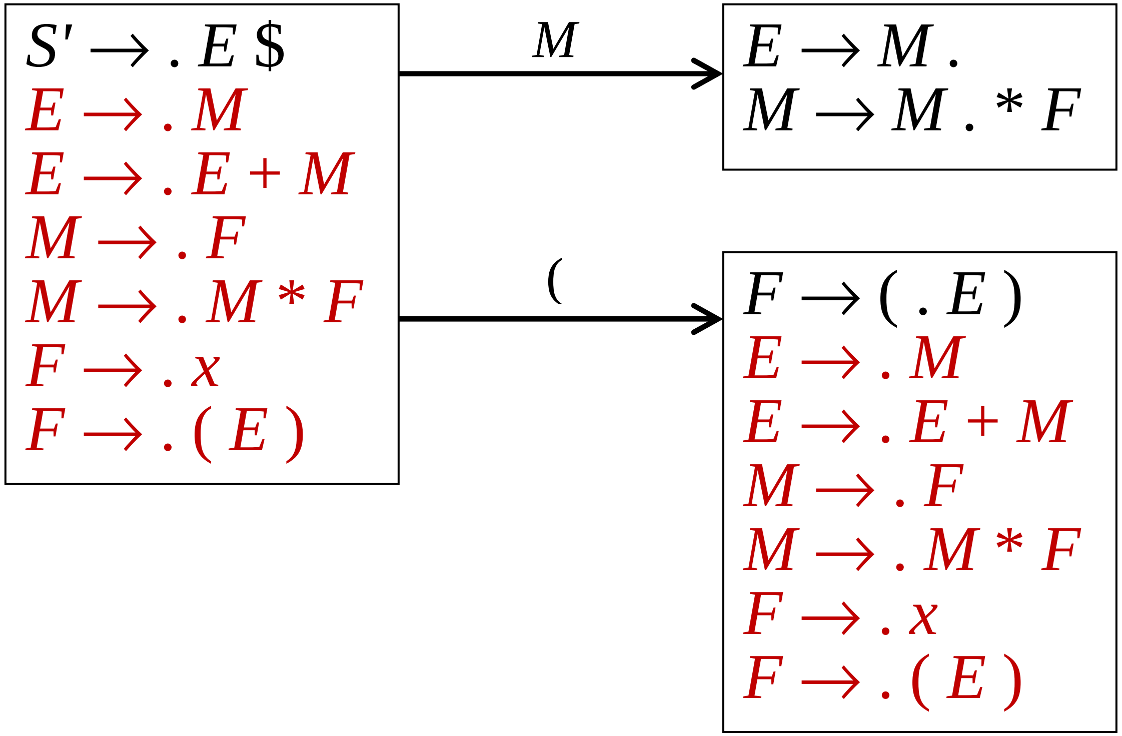
If item $B \rightarrow .\,\beta$ is not in J
then add $B \rightarrow .\,\beta$ to J.

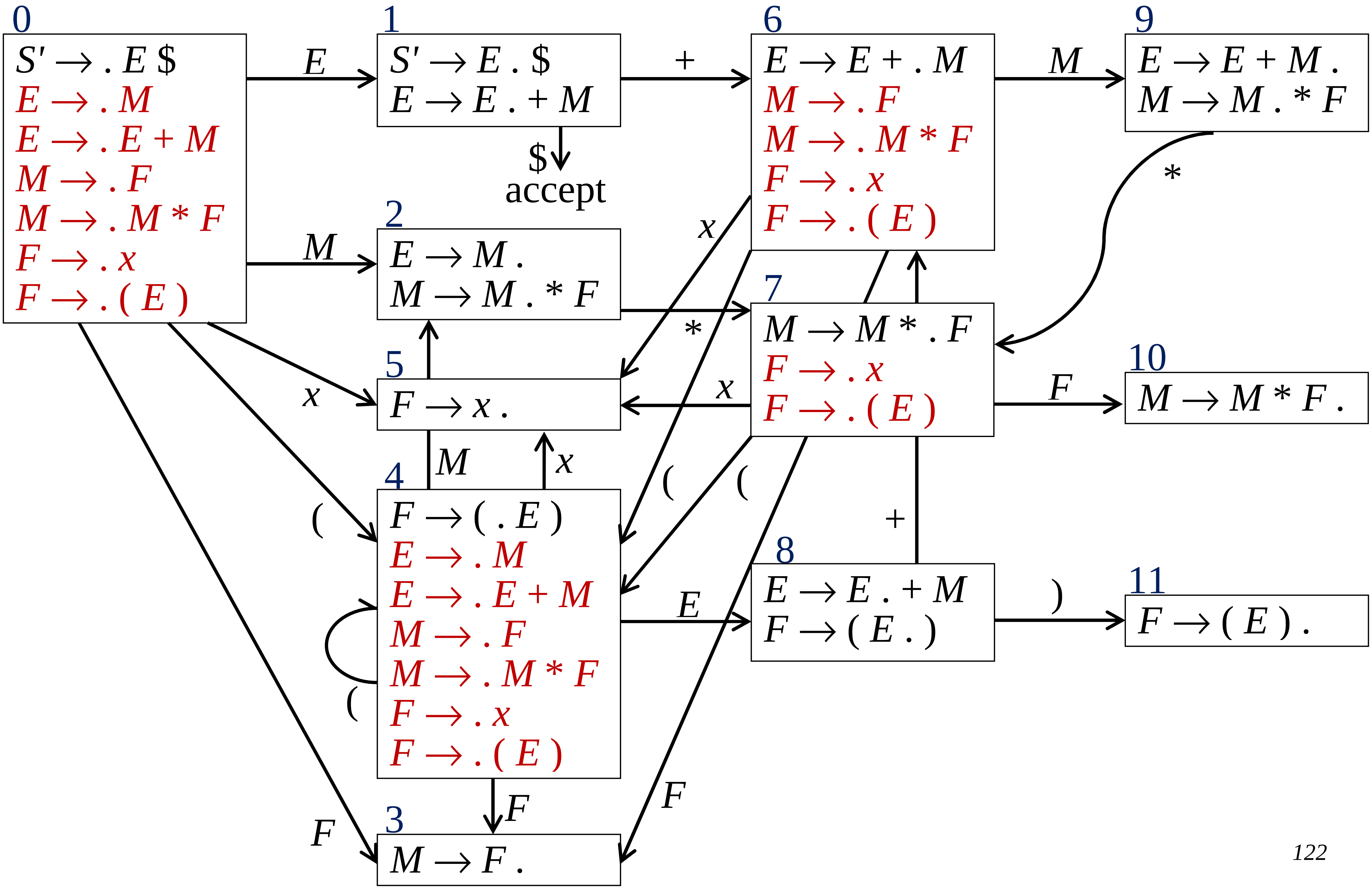
LR(0) automaton construction

If state I contains item $A \to \alpha \,.\, X\, \gamma$ there is:

- A state J containing the closure of item $A \to \alpha\, X \,.\, \gamma$
- An edge from I to J labelled X.

E.g.:




0
S' → . E $
E → . M
E → . E + M
M → . F
M → . M * F
F → . x
F → . (E)
1
S' → E . $
E → E . + M
$
accept
2
E → M .
M → M . * F
3
M → F .
4
F → (. E)
E → . M
E → . E + M
M → . F
M → . M * F
F → . x
F → . (E)
5
F → x .
6
E → E + . M
M → . F
M → . M * F
F → . x
F → . (E)
7
M → M * . F
F → . x
F → . (E)
8
E → E . + M
F → (E .)
9
E → E + M .
M → M . * F
10
M → M * F .
11
F → (E) .
E
M
x
(
F
+
*
)

Corresponding state transition table:

State	*x*	+	*	(	)	$	*E*	*M*	*F*
0	5			4			1	2	3
1		6				acc			
2			7						
3									
4	5			4			8	2	3
5									
6	5			4				9	3
7	5			4					10
8		6			11				
9			7						
10									
11									

LR parsing algorithm

Known as *shift/reduce* parsing.

Basic idea (uses state transition table m):

1. Stack initially contains start state.
2. Current state s is always topmost state on stack.
3. Repeatedly:

   Shift (if s contains "$A \rightarrow \alpha\ .\ a\ \beta$"): read terminal symbol a from input, find new state $s_1 = m[s,a]$, push s_1 on stack.

   Reduce (if s contains "$A \rightarrow \gamma\ .$"): pop $|\gamma|$ states from stack, get current state s from stack, find new state $s_1 = m[s,A]$, push s_1 on stack.
4. Stop (accept) when start symbol is reduced (or when $ shifted).

LR parsing example

Stack	Input	Action	
0	$(x)\$$	shift 4 = $m[0,(]$	
0 4	$x)\$$	shift 5 = $m[4,x]$	
0 4 5	$)\$$	reduce $F \rightarrow x$	pop 5, $m[4,F]$=3, push 3
0 4 3	$)\$$	reduce $M \rightarrow F$	pop 3, $m[4,M]$=2, push 2
0 4 2	$)\$$	reduce $E \rightarrow M$	pop 2, $m[4,E]$=8, push 8
0 4 8	$)\$$	shift 11 = $m[8,)]$	
0 4 8 11	$\$$	reduce $F \rightarrow (E)$	pop 4 8 11, $m[0,F]$=3, push 3
0 3	$\$$	reduce $M \rightarrow F$	pop 3, $m[0,M]$=2, push 2
0 2	$\$$	reduce $E \rightarrow M$	pop 2, $m[0,E]$=1, push 1
0 1	$\$$	shift "accept" = $m[1,\$]$	accept

LR parsing tables

Instead of using transition table, better method is to construct two tables:

- ACTION[s,a]: what to do if terminal symbol a is read in state s.
  - shift s' (if s contains $A \rightarrow \alpha \, . \, a \, \beta$)
  - reduce $A \rightarrow \gamma$ (if s contains $A \rightarrow \gamma \, .$)
  - accept (if s contains $S' \rightarrow S \, .$)
- GOTO[s,A]: state to change to if nonterminal symbol A is reduced in state s.

This simplifies the algorithm.

LR parsing algorithm

```
push start state onto stack;
a = first input symbol;
while (true) {
   s = state on top of stack;
   if (ACTION[s,a] == shift s') {
      push s' onto stack;
      a = next input symbol;
   }
   else if (ACTION[s,a] == reduce A → γ) {
      pop |γ| states from stack;
      s' = state on top of stack;
      push GOTO[s',A] onto stack;
      output production A → γ;
   }
   else if (ACTION[s,a] == accept) {
      break;
   }
   else error();
}
```

LR parsing tables – GOTO

State	*E*	*M*	*F*
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

LR parsing tables – ACTION

State	***x***	**+**	*****	**(**	**)**	**$**
0	s5			s4		
1		s6				accept
2	r1	r1	s7 r1	r1	r1	
3	r3	r3	r3	r3	r3	
4	s5			s4		
5	r5	r5	r5	r5	r5	
6	s5			s4		
7	s5			s4		
8		s6			s11	
9	r2	r2	s7 r2	r2	r2	
10	r4	r4	r4	r4	r4	
11	r6	r6	r6	r6	r6	

Shift/reduce conflicts

- **Problem**: Some table entries contain both shift and reduce actions.
- These are *shift/reduce conflicts*.
- Conflict occurs when a state contains both a shift item ($A \to \alpha \ .\ a\ \beta$) and a reduce item ($A \to \gamma\ .$). E.g., states 2 and 9.
- This indicates that the grammar is not LR(0).
- But it is SLR(1). SLR = “Simple LR”.

SLR(1) parsing

SLR(1) is similar to LR(0) but constructs slightly different tables:

- ACTION[s,a]: what to do if terminal symbol a is read in state s.
  - shift s' if s contains $A \rightarrow \alpha\ .\ a\ \beta$
  - reduce $A \rightarrow \gamma$ if s contains $A \rightarrow \gamma\ .$ and $a \in$ FOLLOW(A)
  - accept if s contains $S' \rightarrow S\ .$
- GOTO[s,A]: state to change to if nonterminal symbol A is reduced in state s.

FOLLOW(E) = { +,), \$ }
FOLLOW(M) = FOLLOW(F) = { *, +,), \$ }

SLR(1) parsing table

State	*x*	+	*	(	)	$
0	s5			s4		
1		s6				accept
2		r1	s7		r1	r1
3		r3	r3		r3	r3
4	s5			s4		
5		r5	r5		r5	r5
6	s5			s4		
7	s5			s4		
8		s6			s11	
9		r2	s7		r2	r2
10		r4	r4		r4	r4
11		r6	r6		r6	r6

More powerful bottom-up parsers

- LR(1): similar to LR(0) but item includes production, position, *and* lookahead symbol. Parsing tables can be very large.
- LALR(1): similar to LR(1) but merges states if only lookahead symbols differ. Parsing tables are smaller. “LookAhead LR(1)”.
- Hierarchy: LR(0) < SLR(1) < LALR(1) < LR(1) < LR(2) < …